Universidad Nacional Mayor de San Marcos

Universidad del Perú. Decana de América

Dirección General de Estudios de Posgrado

Facultad de Ciencias Matemáticas

Unidad de Posgrado

Gestión de subprocesos en la enseñanza-aprendizaje de subprocesos y procesos de los sistemas operativos

TESIS

Para optar el Grado Académico de Magíster en Computación e Informática

AUTOR

Francisco Santiago AGUILAR VÁSQUEZ

ASESOR

Pedro Celso CONTRERAS CHAMORRO

Lima, Perú

2011

Referencia bibliográfica

Aguilar F. (2011). *Gestión de subprocesos en la enseñanza-aprendizaje de subprocesos y procesos de los sistemas operativos*. Tesis para optar el grado de Magíster en Computación e Informática. Unidad de Posgrado, Facultad de Ciencias Matemáticas, Universidad Nacional Mayor de San Marcos, Lima, Perú.

RESUMEN

GESTION DE SUBPROCESOS EN LA ENSEÑANZA – APRENDIZAJE DE SUBPROCESOS Y PROCESOS DE LOS SISTEMAS OPERATIVOS

Francisco Santiago Aguilar Vásquez

ASESOR : Dr. Pedro Celso Contreras Chamorro
Grado obtenido : Magister en Computación e Informática

Mientras que para su enseñanza-aprendizaje en sistemas operativos, los procesos y subprocesos se describen con lenguajes de programación general; para su ejecución en el computador, estos mismos procesos y subprocesos se expresan en lenguaje máquina. La distancia entre los lenguajes de descripción y ejecución de procesos y subprocesos, así como la diferencia entre las velocidades de su ejecución y de su enseñanza-aprendizaje afectan este proceso de enseñanza-aprendizaje.

En este trabajo se desarrolla un sistema software "Gestor Académico de subprocesos (GASP)" que permite al usuario controlar la velocidad de ejecución de los subprocesos y viabiliza a los subprocesos presentar su avance de ejecución en el lenguaje de descripción. El GASP es un emulador limitado del núcleo de un sistema operativo multitarea. Se presenta la deducción de requisitos y su análisis, así como el diseño y la implementación. Los requisitos funcionales reflejan los servicios del núcleo del sistema operativo. Entre los requisitos no funcionales se consideran aspectos planteados por las teorías del conocimiento.

Para fines de prueba, se ha construido un conjunto de subprocesos demostrativos que se integran con el GASP por medio de una interfaz gráfica (IGASP) amigable al usuario, resultando en una aplicación desktop (sysGASP).

PALABRAS CLAVES:

ENSEÑANZA-APRENDIZAJE
SISTEMA OPERATIVO
PROCESO Y SUBPROCESO
VELOCIDAD DE EJECUCIÓN DE SUBPROCESOS
LENGUAJE DE PROGRAMACIÓN
PROCESO DE DESARROLLO DE SOFTWARE
APLICACIÓN DESKTOP.

ABSTRACT

THREAD MANAGEMENT IN TEACHING – LEARNING THREADS AND PROCESSES OF OPERATING SYSTEMS

Francisco Santiago Aguilar Vásquez

ADVISOR : Pedro Celso Contreras Chamorro, Ph. D.
Given degree : Master of Science in Computing and Information Tecnology

---

While for teaching/learning in operating systems, processes and threads are described in general programming languages, to run on the computer, these same processes and threads are coded in machine language. The distance between the languages of description and execution of processes and threads, and the difference between the speeds of its execution and its teaching/learning affect this teaching/learning process.

This work develops a software system "thread Academic Manager (GASP)" that allows the user to control the speed of execution of threads and facilitates threads to show progress in the language of description. GASP is a limited emulated core of a multitasking operating system. Requirements are elicited and analyzed concurrently with design and implementation of the system. Functional requirements reflect the core services of the operating system. Among the non-functional requirements are considered issues raised by the theories of knowledge.

For testing purposes, it has built a demonstration set of threads that integrate with the GASP through a user friendly GUI (IGASP), resulting in a desktop application (sysGASP).

KEYWORDS:

TEACHING/LEARNING
OPERATING SYSTEM
PROCESSES AND THREADS
THREAD EXECUTION SPEED
PROGRAMMING LANGUAGE
SOFTWARE DEVELOPMENT PROCESS
DESKTOP APPLICATION

GESTION DE SUBPROCESOS EN LA ENSEÑANZA-APRENDIZAJE DE SUBPROCESOS Y PROCESOS DE LOS SISTEMAS OPERATIVOS.

Francisco Santiago Aguilar Vásquez.

Tesis presentada a consideración del Jurado Examinador nombrado por la Unidad de Post Grado de la Facultad de Ciencias Matemáticas de la Universidad Nacional Mayor de San Marcos, como parte de los requisitos para obtener el grado académico de Magister en Computación e Informática.

Aprobada por:

_____	_____
Mg. Jorge Icaro Condado Jáuregui **Presidente**	Mg. Augusto Parcemón Cortez Vásquez **Miembro**
_____	_____
Mg. Virginia Vera Pomalaza **Miembro**	Mg. Jorge Aurelio Rodríguez Huerta **Miembro**

_____

Dr. Pedro Celso Contreras Chamorro
Miembro Asesor

Lima – Perú

2011

FICHA CATALOGRAFICA

AGUILAR VASQUEZ, FRANCISCO SANTIAGO

GESTIÓN DE SUBPROCESOS EN LA ENSEÑANZA – APRENDIZAJE DE SUBPROCESOS Y PROCESOS DE LOS SISTEMAS OPERATIVOS, (LIMA) 2011.

xi, 189 p. 29.7 cm. (UNMSM, Magister, Computación e Informática, 2011).
Tesis, Universidad Nacional Mayor de San Marcos,
Facultad de Ciencias Matemáticas
Unidad de Post Grado
Computación e Informática
UNMSM / FdeCM

RESUMEN

GESTION DE SUBPROCESOS EN LA ENSEÑANZA – APRENDIZAJE DE SUBPROCESOS Y PROCESOS DE LOS SISTEMAS OPERATIVOS

Francisco Santiago Aguilar Vásquez

ASESOR : Dr. Pedro Celso Contreras Chamorro
Grado obtenido : Magister en Computación e Informática

Mientras que para su enseñanza-aprendizaje en sistemas operativos, los procesos y subprocesos se describen con lenguajes de programación general; para su ejecución en el computador, estos mismos procesos y subprocesos se expresan en lenguaje máquina. La distancia entre los lenguajes de descripción y ejecución de procesos y subprocesos, así como la diferencia entre las velocidades de su ejecución y de su enseñanza-aprendizaje afectan este proceso de enseñanza-aprendizaje.

En este trabajo se desarrolla un sistema software "Gestor Académico de subprocesos (GASP)" que permite al usuario controlar la velocidad de ejecución de los subprocesos y viabiliza a los subprocesos presentar su avance de ejecución en el lenguaje de descripción. El GASP es un emulador limitado del núcleo de un sistema operativo multitarea. Se presenta la deducción de requisitos y su análisis, así como el diseño y la implementación. Los requisitos funcionales reflejan los servicios del núcleo del sistema operativo. Entre los requisitos no funcionales se consideran aspectos planteados por las teorías del conocimiento.

Para fines de prueba, se ha construido un conjunto de subprocesos demostrativos que se integran con el GASP por medio de una interfaz gráfica (IGASP) amigable al usuario, resultando en una aplicación desktop (sysGASP).

PALABRAS CLAVES:

ENSEÑANZA-APRENDIZAJE
SISTEMA OPERATIVO
PROCESO Y SUBPROCESO
VELOCIDAD DE EJECUCIÓN DE SUBPROCESOS
LENGUAJE DE PROGRAMACIÓN
PROCESO DE DESARROLLO DE SOFTWARE
APLICACIÓN DESKTOP.

ABSTRACT

THREAD MANAGEMENT IN TEACHING – LEARNING THREADS AND PROCESSES OF OPERATING SYSTEMS

Francisco Santiago Aguilar Vásquez

ADVISOR : Pedro Celso Contreras Chamorro, Ph. D.
Given degree : Master of Science in Computing and Information Tecnology

---

While for teaching/learning in operating systems, processes and threads are described in general programming languages, to run on the computer, these same processes and threads are coded in machine language. The distance between the languages of description and execution of processes and threads, and the difference between the speeds of its execution and its teaching/learning affect this teaching/learning process.

This work develops a software system "thread Academic Manager (GASP)" that allows the user to control the speed of execution of threads and facilitates threads to show progress in the language of description. GASP is a limited emulated core of a multitasking operating system. Requirements are elicited and analyzed concurrently with design and implementation of the system. Functional requirements reflect the core services of the operating system. Among the non-functional requirements are considered issues raised by the theories of knowledge.

For testing purposes, it has built a demonstration set of threads that integrate with the GASP through a user friendly GUI (IGASP), resulting in a desktop application (sysGASP).

KEYWORDS:

TEACHING/LEARNING
OPERATING SYSTEM
PROCESSES AND THREADS
THREAD EXECUTION SPEED
PROGRAMMING LANGUAGE
SOFTWARE DEVELOPMENT PROCESS
DESKTOP APPLICATION

INDICE

INTRODUCCION

CAPITULO I

NOCIONES DEL PROCESO DE ENSEÑANZA-APRENDIZAJE

1.1 Principales teorías del aprendizaje 5
1.1.1 Teorías conductistas 5
1.1.1.1 Condicionamiento clásico 6
1.1.1.2 Condicionamiento operante 6
1.1.1.3 Condicionamiento observacional 7
1.1.2 Teorías cognitivas 8
1.1.2.1 Teoría del procesamiento de información 9
1.1.3 Teorías socioculturales 11
1.2 Elementos del proceso de enseñanza-aprendizaje 13
1.2.1 Contexto 14
1.2.2 Docente 14
1.2.3 Estudiante 15
1.2.4 Estrategias metodológicas 15
1.2.5 Contenido del aprendizaje 16

CAPITULO II

EL COMPUTADOR

2.1 Arquitectura hardware del computador 17
2.1.1 El procesador 19
2.1.1.1 Unidad de procesos 19
2.1.1.2 Unidad de control 21
2.1.2 Memoria 22
2.1.3 Periféricos e interfaz de E/S 23
2.1.3.1 Interfaces de entrada/salida 25
2.1.3.2 Entrada/salida controlada por programa 28

2.1.3.3 Entrada/salida guiada por interrupciones 30
2.1.4 Repertorio de instrucciones 31
2.2 Sistema operativo 32
2.2.1 Tipos 34
2.2.1.1 Sistemas operativos serie 35
2.2.1.2 Sistemas operativos de lotes 35
2.2.1.3 Sistemas operativos de multiprogramación 36
2.2.1.3.1 Sistemas operativos de tiempo compartido 37
2.2.1.3.2 Sistemas operativos de tiempo real 38
2.2.1.3.3 Sistema operativos combinados 39
2.2.1.4 Sistemas operativos distribuidos 40
2.2.2 Perspectivas 40
2.2.2.1 Perspectiva del usuario de lenguaje de órdenes 41
2.2.2.2 Perspectiva del usuario de llamadas al sistema 43
2.2.3 Funciones básicas 44
2.2.3.1 Gestión de procesos, hilos y recursos 45
2.2.3.2 Gestión de memoria 46
2.2.3.3 Gestión de dispositivos 48
2.2.3.4 Gestión de archivos 50
2.2.4 Arquitectura 52

CAPITULO III

GESTION DE PROCESOS Y SUBPROCESOS

3.1 Procesos, subprocesos y recursos 57
3.1.1 Máquinas abstractas 57
3.1.2 Servicios 61
3.1.3 Proceso 62
3.1.4 Subproceso 63
3.1.5 Diagrama de estados 64
3.1.6 Recursos 65
3.2 Planificación 67
3.2.1 Mecanismo de planificación 69
3.2.2 Criterios de selección y rendimiento 71

3.2.3 Estrategias no expropiativas 73
3.2.4 Estrategias expropiativas 74
3.3 Sincronización 75
3.3.1 Sección crítica y exclusión mutua 76
3.3.2 Semáforos 79
3.3.3 Soporte hardware de semáforos 82
3.3.4 Semáforos con colas 84
3.3.5 Abstracciones de semáforos 84
3.3.5.1 Sincronización simultánea 85
3.3.5.2 Eventos 86
3.3.5.3 Monitores 87
3.4 Comunicación 90
3.4.1 Mensajes 91
3.4.2 Comunicación y sincronización por mensajes 94
3.4.3 Señalización de interrupciones por mensajes 96
3.5 Interbloqueo 97
3.5.1 Prevención 98
3.5.2 Evitación 99
3.5.3 Detección y recuperación 100

CAPITULO IV

GESTOR ACADEMICO DE SUBPROCESOS

4.1 Definición de servicios del GASP 102
4.1.1 Servicios 102
4.1.2 Soporte de servicios 103
4.1.3 Transiciones de estado de los subprocesos 105
4.2 Factor académico 107
4.2.1 Lenguaje de programación 107
4.2.2 Velocidad de ejecución de programas 109
4.3 Interfaz 110
4.3.1 Interfaz de usuario final 110
4.3.2 Interfaz con los subprocesos 111
4.3.3 Otras restricciones 111

4.4 Especificación funcional 111
4.4.1 Planificación básica de subprocesos 112
4.4.2 Sincronización y comunicación de subprocesos 112
4.4.3 Gestión de interrupciones 113
4.4.4 Administración de subprocesos 114
4.4.5 Configuración inicial 116

CAPITULO V

DISEÑO E IMPLEMENTACION DEL GASP

5.1 Consideraciones de diseño e implementación 117
5.1.1 Elementos de hardware emulados 117
5.1.2 Control de velocidad de ejecución de SPP 118
5.1.3 Acreditación de ejecución referencial de SPP 119
5.1.4 Conmutación de SPP 119
5.2 Arquitectura 120
5.2.1 Contexto 120
5.2.2 Capas estructurales 120
5.2.3 Paquetes de clases 121
5.3 Clases y objetos 123
5.3.1 Máscara de interrupciones 123
5.3.2 Emulador hardware 124
5.3.3 Elementos básicos 128
5.3.4 Listas de elementos básicos 133
5.3.5 Núcleo 138
5.3.5.1 Configuración inicial 139
5.3.5.2 Gestión de buzones 140
5.3.5.3 Planificación básica de subprocesos 140
5.3.5.4 Interacción entre subprocesos 140
5.3.5.5 Gestión de interrupciones 141
5.3.5.6 Administración de subprocesos 141
5.3.6 Igestor 142
5.3.7 Tareas específicas 151
5.4 Interfaz de integración del GASP 151

5.4.1 Interfaz de preparación de subproceso 154
5.5 Arquitectura de implementación y despliegue 155
5.6 Componentes del GASP 156
5.6.1 Máscara de interrupciones 157
5.6.2 Emulador hardware 157
5.6.3 Elementos básicos 158
5.6.4 Listas de elementos básicos 158
5.6.5 Núcleo 159
5.6.6 Interfaz del GASP 159
5.6.7 Tareas específicas 160
5.7 Componentes de la interfaz de integración 161

CAPITULO VI
SUBPROCESOS GESTIONADOS EN EL GASP

6.1 Estructura del SPG 162
6.1.1 Arquitectura del subproceso 162
6.1.2 Interfaz de usuario del subproceso 163
6.1.3 Lógica de aplicación del subproceso 165
6.2 Construcción del SPG 166
6.2.1 Diseño de la estructura 166
6.2.2 Componentes de producción 167
6.2.3 Código fuente en C# 168
6.3 Casos demostrativos de SPGs en el GASP 174
6.3.1 Subprocesos independientes 175
6.3.2 Subprocesos colaborativos 179
6.3.3 Subprocesos productores y consumidores 181

CAPITULO VII
CONCLUSIONES Y SUGERENCIAS

7.1 Conclusiones 185
7.2 Sugerencias 188

LISTA DE CUADROS 190

LISTA DE FIGURAS 191

BIBLIOGRAFIA 194

ANEXOS

LISTADOS DE PROGRAMAS DEL sisGASP

A. Fuentes del GASP en C# A1

B. Fuentes de la IGASP en C# B1

C. Fuentes de subProcesos gestionados en C# C1

INTRODUCCION.

El computador está omnipresente en la sociedad actual, tanto en forma directa como dentro de un diversidad de sistemas que van desde un celular hasta el satélite, desde un juguete hasta sistemas educativos reales y virtuales, desde un identificador de artículos hasta gestores de inventarios y producción, desde un controlador de pistola que dispara trozos de corcho hasta complejos sistemas de guerra que podrían terminar con la humanidad… Las personas se preocupan por obtener computadores de última tecnología, sin haber explorado y conocido muchas facilidades que brindan los que se abandonan. El computador se ha convertido en un objeto de conocimiento que exige conocerlo, y junto con los sistemas que lo incorporan, también se convierten en medios y recursos de enseñanza – aprendizaje en general.

Como contenido de aprendizaje, la naturaleza del computador plantea el hardware, constituido por componentes físicos (memoria, procesador, interfaces de entrada/salida y periféricos), y el software, compuesto por programas y datos. El software básico que administra el hardware y media entre este hardware y el resto del software (los programas aplicativos), incluyendo el usuario (operador) se denomina sistema operativo. Dentro del sistema operativo, la gestión de procesos y subprocesos se encarga de administrar el procesador y otros recursos para ejecutar programas.

La enseñanza – aprendizaje de la gestión de procesos y subprocesos experimenta dificultades, entre otras cosas, por la diferencia entre la velocidad de ejecución de programas y la velocidad de enseñanza – aprendizaje viable, así como porque el lenguaje en el cual se ejecutan los programas es diferente al lenguaje en el que se conciben.

Enseñanza – aprendizaje de Procesos y subprocesos de los Sistemas Operativos.

Los sistemas operativos son programas básicos de computadores que constituyen contenidos de aprendizaje que suelen organizarse como temas o asignaturas teórico – experimentales (prácticas) en los currículos de formación técnica, universitaria o de postgrado en las áreas de computación. La gestión de procesos y subprocesos destacan, por su importancia, como sendas unidades temáticas dentro de estas asignaturas.

La enseñanza – aprendizaje de los procesos y subprocesos de los sistemas operativos (EAPSO) plantea conocimientos previos sobre computación al estudiante, competencias didácticas y profesionales sobre sistemas operativos al docente, contenidos significativamente organizados sobre el tema y estrategias didácticas específicas dentro de un contexto espacial y temporal con medios y recursos apropiados, con el cual el estudiante interactúa para construir sus conocimientos sobre procesos y subprocesos de los sistemas operativos, facilitado/mediado por el docente.

Existe una inmensa variedad y cantidad de información en diferentes medios (papel, electrónico, magnético, óptico) y en diferentes formas (texto, gráficos, audio, imágenes, video), así como una gran diversidad de medios y recursos que permiten disponer de esta información (libros, revistas, periódicos, tesis, estudios, actas, computadores, programas, lenguajes de programación, librerías, redes, ambientes objetivos, ambientes virtuales, sistemas administrativos), para estructurar los contenidos y organizar las estrategias metodológicas para EAPSO.

La EAPSO en el escenario descrito implica, entre otras cosas, una velocidad de enseñanza compatible con la velocidad de aprendizaje del estudiante y una organización significativa de la información sobre procesos y subprocesos de sistemas operativos. La significatividad de estos contenidos y las velocidades de su facilitación y percepción, por el lado teórico del tema, son viables para los docentes y estudiantes nacionales, dada la disponibilidad de la información accesible en sus idiomas y lenguajes y a sus propias velocidades. Sin embargo esta significatividad y estas velocidades, por el lado experimental del tema, sufren dificultades.

La significatividad tiene dificultades, esencialmente, porque los programas que se usan para experimentar estas entidades computacionales se expresan por docentes y estudiantes en lenguajes de programación general (C++, java, C#) y se ejecutan por el computador en lenguaje máquina. En la figura 4.2 se presenta una aproximación de esta significatividad, proyectada como nivel de comprensión, en función de la distancia entre el lenguaje de programación y el lenguaje de ejecución de programas. El nivel de comprensión es mayor, mientras menor sea la distancia entre estos lenguajes.

Los computadores vigentes ejecutan los programas a centenas de millones de instrucciones de lenguaje máquina por segundo, velocidades que dificultan la facilitación y percepción para la enseñanza - aprendizaje. En la figura 4.3 se presenta una aproximación de velocidades deseables de ejecución de programas en lenguajes de

programación general para su facilitación y percepción durante la enseñanza - aprendizaje, en términos de nivel de comprensión versus logaritmo del número de instrucciones por segundo. La velocidad deseable ronda alrededor de una instrucción en lenguaje de programación general por segundo.

Herramienta - software para la enseñanza - aprendizaje de subprocesos y procesos de los Sistemas Operativos.

Este trabajo define y desarrolla un gestor académico de subprocesos para la enseñanza – aprendizaje de subprocesos y procesos de los sistemas operativos, que gestiona subprocesos con dinámicas de ejecución controladas por el usuario (docente o estudiante), y que muestra sus avances de ejecución en el lenguaje fuente. Este gestor adapta la velocidad de ejecución de subprocesos a la velocidad de facilitación del docente y de percepción del estudiante y el avance de ejecución de los subprocesos lo presenta en un lenguaje de programación general. Así mismo, el trabajo integra en la herramienta ejemplos de subprocesos pertinentes.

El gestor es una aplicación desktop que se desarrolla siguiendo un enfoque orientado a objetos, empleando UML para el modelamiento y ms vs.net con C# para la implementación. La aplicación es implantable en plataformas .NET.

Organización del Informe.

Además de la presente introducción, el informe del presente trabajo comprende siete capítulos y anexos con los contenidos que a continuación se indican.

El capítulo I presenta aspectos didácticos, tales como las teorías vigentes de aprendizaje y los elementos que constituyen el proceso de enseñanza – aprendizaje.

El capítulo II describe el hardware y el software del computador. El hardware se interpreta dentro de la arquitectura del modelo Von Newman del hardware, incluyendo: procesador, memoria, interfaces de entrada/salida, periféricos y sistemas de intercomunicaciones. Dentro del software, se presenta un sistema operativo multitarea básico en sus perspectivas de interfaz y administrador de recursos.

El capítulo III explora la gestión de procesos y subprocesos de un sistema operativo básico multitarea. Se describen con suficiente precisión: los procesos, subprocesos y recursos como mecanismos de ejecución de programas; las estrategias de planificación y los estados de procesos/subprocesos, la sincronización en la

colaboración de procesos/subprocesos, la comunicación entre procesos/subprocesos y los interbloqueos entres procesos/subprocesos.

El capítulo IV concibe el gestor académico de subprocesos (GASP), como herramienta software, estableciendo sus requisitos funcionales y no funcionales. Se definen los servicios que el GASP brinda y las transiciones de estado de los subprocesos. Dentro del factor académico se resuelve la diferencia entre los lenguajes de programación y ejecución y se incorpora como requisito del GASP el control de la dinámica de ejecución de los subprocesos. Se caracterizan las interfaces del GASP con el usuario y con los subprocesos. Finalmente se presenta la especificación funcional del GASP.

El capítulo V diseña e implementa el GASP y la interfaz gráfica de integración (IGASP). Precisa los requisitos, especialmente los no funcionales. El diseño muestra la arquitectura del GASP por capas en su contexto y los paquetes por capa. Para cada paquete, presenta el diseño de clases y objetos. La implementación muestra la arquitectura de implementación y despliegue del GASP en su contexto. La implementación del GASP y de la interfaz de integración presenta sus componentes de producción y despliegue. Los primeros organizados por paquetes.

El capítulo VI incluye la descripción del desarrollo de los subprocesos de ejecución controlada y casos de prueba del GASP en su contexto. El desarrollo de los subprocesos contempla su arquitectura a nivel diseño y sus componentes a nivel de implementación, en relación con el GASP y la interfaz de integración. Los casos de prueba consideran diversos escenarios de prueba del GASP empleando diversos subproceso implementados sobre esta plataforma.

El capítulo VII contiene las conclusiones y sugerencias.

Los anexos incluyen el código fuente en lenguaje orientado a objetos de programación general C#, organizados en tres partes: Fuentes del GASP, Fuentes de la IGASP y Fuentes de subProcesos gestionados.

Capitulo I:

NOCIONES DEL PROCESO DE ENSEÑANZA-APRENDIZAJE.

A partir de los aportes de la Psicología, la Teoría General de Sistemas y la Cibernética, la Teoría de las Comunicaciones, la Computación, la Sociología, la Antropología, la Filosofía y otras ciencias, las corrientes pedagógicas contemporáneas cubren aspectos tan variados como complejos para promover el proceso de enseñanza – aprendizaje. Entre estos aspectos se encuentran la fundamentación, la planificación, el diseño, la programación y la ejecución de los procesos de enseñanza; la producción de materiales educativos; la innovación de medios de comunicación escolarizada y no escolarizada; la adopción de la tecnología de la información y las comunicaciones en la enseñanza – aprendizaje virtual. Estos temas son tratados desde diferentes perspectivas por diversidad de autores, incluyendo [1], [2], [7], [11], [18] y [19] que se toman como base para aproximarse a la enseñanza – aprendizaje de los sistemas operativos.

En la aproximación a la enseñanza – aprendizaje de los sistemas operativos, este capítulo presenta: las principales teorías vigentes del aprendizaje y los elementos del proceso enseñanza - aprendizaje.

1.1 PRINCIPALES TEORIAS DEL APRENDIZAJE.

El proceso de enseñanza – aprendizaje normalmente es guiado por una teoría de aprendizaje o una combinación de ellas. La teoría del aprendizaje explica como ocurre el aprendizaje. Las principales teorías del aprendizaje, vigentes en el presente siglo, se presentan en tres grupos: conductistas, cognitivas y socioculturales.

1.1.1 TEORIAS CONDUCTISTAS.

Estas teorías se desarrollan principalmente en la primera mitad del siglo anterior y se centran en conductas objetivas y observables, descartando las actividades mentales.

El aprendizaje se define como la adquisición de nuevas conductas o comportamientos y es generado por un procedimiento experimental denominado condicionamiento, manejado por estímulos ambientales

Según Beltrán (1987), citado por [7], el modelo del condicionamiento puede describirse de la siguiente manera:

- Se establece una conducta deseada en términos observables y medibles.
- Se selecciona un sistema concreto para medir la conducta.
- Ser valora el nivel de conducta en el punto de partida.
- Se prepara un programa para conseguir la conducta deseada.
- Se selecciona las actividades que facilitarán la aparición de la conducta deseada.
- Se condiciona según el programa y sus actividades seleccionadas.
- Se evalúa la efectividad el programa.

Entre estas teorías se plantean las siguientes variantes: condicionamiento clásico, condicionamiento instrumental u operante y condicionamiento observacional o imitativo.

1.1.1.1 Condicionamiento clásico.

Iván Pavlov, Vladimir Béchterev, Iván Séchenov y John Watson, citados, son autores de esta teoría que define la ley de asociación o contigüidad o conexión en el aprendizaje: si se asocia, varias veces, un estímulo neutro con un estímulo provocador; el estímulo previamente neutro llegará a provocar la misma clase de respuesta, que inicialmente provocaba el estímulo provocador.

Esta teoría centra su atención en el estímulo que causa la respuesta como se muestra en el modelo E-R de la figura 1.1, es decir en el mecanismo neural de conexión.

Figura 1.1 Modelo de condicionamiento clásico

Al conocer el proceso de condicionamiento del aprendizaje, el docente podrá comprender ciertas conductas y actitudes de los estudiantes frente al proceso de enseñanza – aprendizaje y moldearlas para que permitan un aprendizaje más efectivo.

1.1.1.2 Condicionamiento operante.

Edward Thorndike (1974-1949) y Barrhus Skinner(1904-1990), citados, son autores del condicionamiento operante que persigue la consolidación de la respuesta

según el estímulo, buscando los reforzadores necesarios para implantar esta relación en el individuo.

Esta teoría se centra en la conducta (operante) misma, más que en el mecanismo neural en que se apoya. La conducta operante es aquella que opera sobre el ambiente, no incluye la conducta reflexiva.

Skinner explica el aprendizaje por medio de la conducta operante, en la que participan tres elementos, tal como se aprecia en el modelo de la figura 1.2,. El primero es la ocasión en que la respuesta ocurre, esta ocasión podría ser un estímulo discriminativo (Ed) visible o no necesariamente visible. El Segundo elemento es la respuesta (Rc) que ocurre. El tercer elemento está constituido por las contingencias de reforzamiento (Er) o consecuencias, las cuales forman la relación entre el estímulo y la respuesta. Las consecuencias solamente ocurren si la respuesta es emitida en presencia del estímulo discriminativo.

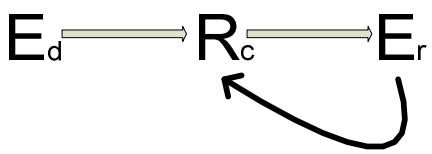


Figura 1.2 Modelo de condicionamiento operante

El reforzamiento de una conducta puede ser positivo, negativo o castigo. El positivo aumenta la probabilidad de la conducta. El negativo ocurre cuando la conducta elimina o evita un estímulo desagradable. El castigo es la consecuencia desagradable o la prohibición de una cosa deseada. Las conductas pueden ser moldeadas, encadenadas, diferenciadas, extinguidas. Los estímulos pueden ser generalizados y discriminados.

La enseñanza se plantea como un programa de contingencias de refuerzos que modifiquen la conducta. Las técnicas de aprendizaje programado, fundamento de las máquinas de enseñar, se basan en esta teoría.

1.1.1.3 Condicionamiento observacional.

Plantea un aprendizaje, investigado por Alberto Bandura en la década de los 80, basado en una situación social en la que, al menos, participan dos personas: el modelo que muestra una conducta y el sujeto que observa esa conducta, esta observación determina el aprendizaje (del sujeto). Este aprendizaje es consecuencia de condicionamientos y procesos cognitivos.

El aprendizaje observacional se realiza en dos fases: adquisición de la conducta y su ejecución. Los procesos de atención y retención se dan en la adquisición de la conducta. En la ejecución de la conducta tienen lugar la reproducción motora y la motivación y refuerzo.

1.1.2 TEORIAS COGNITIVAS.

Estas teorías asumen el aprendizaje como una representación de la realidad producida a partir de la experiencia y ponen énfasis en como estas representaciones son adquiridas, almacenadas y recuperadas de la estructura cognitiva o memoria - valor constructivista. No se niega la existencia de otras formas de aprendizaje inferior; pero si su relevancia, atribuyendo el aprendizaje a procesos constructivos de asimilación y acomodación.

Aspectos comunes planteados por estas teorías se esquematizan en el modelo Estímulo – Organismo – Respuesta de la figura 1.3, donde el sujeto (O) es un procesador activo de la información por medio del registro y la organización para alcanzar la reorganización y reestructuración de la estructura cognitiva. Esta reestructuración no es solamente asimilación, sino construcción y cambio dinámicos del conocimiento.

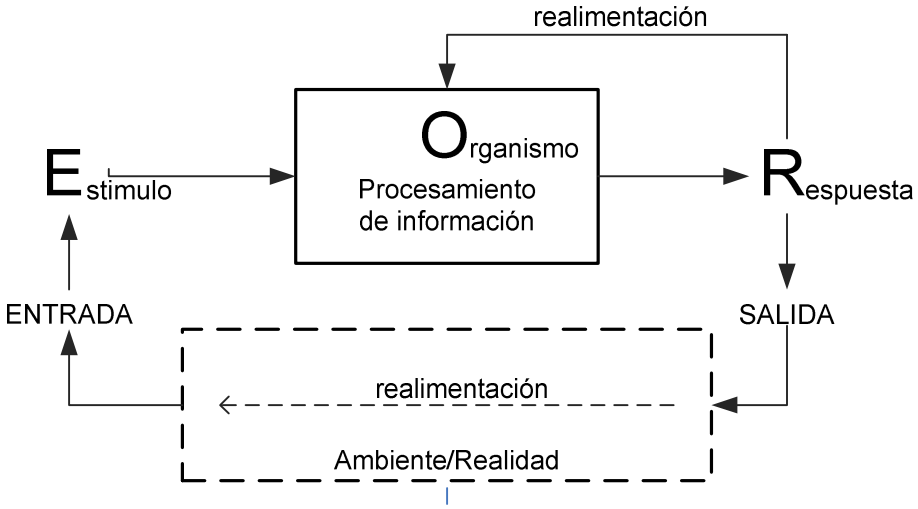


Figura 1.3 Modelo de aprendizaje E-O-R

En el modelo, al recibirse un estímulo del ambiente externo y ser procesado en el interior del organismo, se emite una respuesta (conducta) específica (E→O→R). La respuesta es realimentada directamente al organismo (R→O) para consolidar la

conducta total final. La conducta opera sobre el ambiente, cambiándolo a una nueva situación que se realimenta al organismo como un evento – estimulante.

Durante el siglo anterior, se han desarrollado diferentes teorías cognitivas para explicar lo que ocurre en el interior del organismo durante el aprendizaje, citados, entre las que se destacan:

- Organización activa del campo perceptual de la escuela Gestalt (1925, 1935),
- Conexión entre expectativas y significados de Edward Tolman (1932, 1949),
- Activación de variables intervinientes de Clark Hull (1943),
- Desarrollo cognitivo relacionado con maduración biológica de Jean Piaget (1956),
- Aprendizaje por descubrimiento de Jerome Bruner (1966),
- Procesamiento de información de Robert Gagné (1974 – 1985),
- Zona de desarrollo próximo de Lev Vygotsky (1926),
- Aprendizaje significativo de David Ausubel (1978),
- Acciones mentales de Piotr Galperin (1978).

1.1.2.1 Teoría del procesamiento de información.

Esta teoría, planteada por Gagné (1974), citado, desarrolla un aprendizaje taxonómico, como propuesta ecléctica entre el conductismo, el cognitivismo y el procesamiento de información.

Lo esencial de la teoría se describe en términos de procesos de aprendizaje, fases de aprendizaje, resultados de aprendizaje y condiciones de aprendizaje.

Los procesos de aprendizaje consisten en el cambio de una capacidad (humana), que persiste en el tiempo y no es atribuible al proceso de maduración. El cambio se manifiesta en la conducta, permitiendo inferir que se logró por el aprendizaje.

El modelo de procesamiento de la información, tal como se muestra en la figura 1.4, presenta estructuras para explicar lo que sucede internamente, en el individuo, durante el proceso de aprendizaje.

Vía los receptores (órganos sensoriales) la información llega al registro sensorial, donde las percepciones y eventos son codificados. Esta información pasa a la memoria operativa donde es codificada en forma conceptual.

En este punto se pueden presentar varias alternativas de proceso para su almacenamiento o no en la memoria a largo plazo.

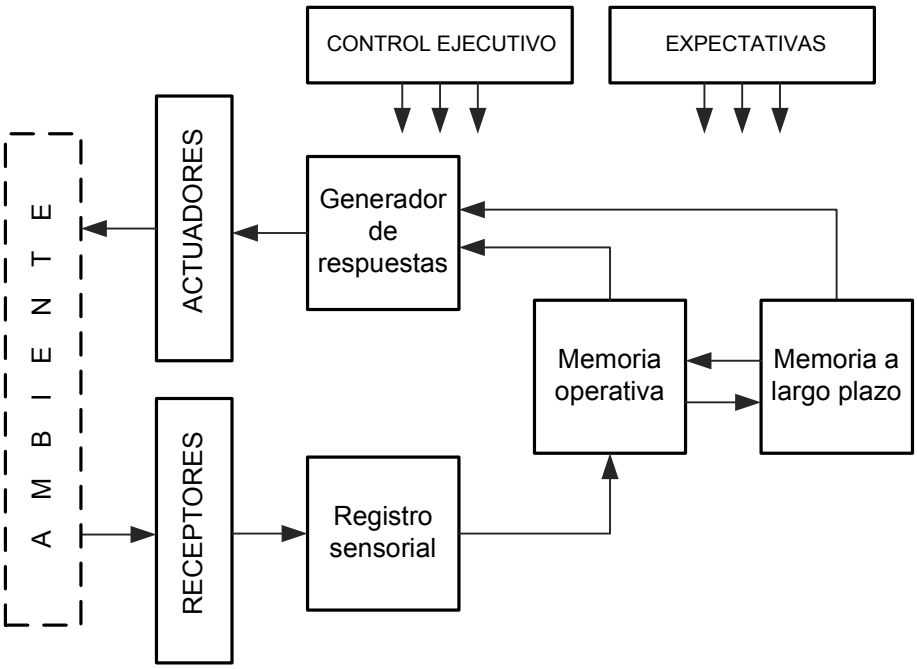


Figura 1.4 Modelo de teorías de procesamiento de información de Gagné.

Una vez que la información ha sido registrada en cualquiera de las dos memorias, ésta puede ser retirada o recuperada, en función de estímulos externos que lo hacen necesaria, pasando al generador de respuestas, que transforma la información en acción, la cual pasa a través de los actuadores hacia el ambiente.

El control ejecutivo y las expectativas son elementos de motivación tanto intrínseca como extrínseca que preparan o estimulan al individuo para codificar y decodificar la información.

Las fases de aprendizaje indican las etapas que suceden durante el acto de aprender:

- Motivación: expectativas,
- Aprehensión: atención perceptiva selectiva,
- Adquisición: codificación y almacenaje,
- Retención: acumulación en la memoria,
- Recuperación: recuperación,
- Generalización: transferencia,
- Desempeño: generación de respuestas y
- Retroalimentación: reforzamiento.

Los procesos y las fases dan lugar a los siguientes tipos de aprendizaje:

- Aprendizaje de señales, equivalente a condicionamiento clásico (reflejos),
- Aprendizaje E-R, equivalente a condicionamiento instrumental u operante,
- Encadenamiento motor,
- Asociación verbal, E-R en el área verbal,
- Discriminaciones múltiples,
- Aprendizaje de conceptos: concretos y abstractos,
- Aprendizaje de principios y reglas y
- Aprendizaje de solución de problemas.

Los resultados de aprendizaje señalan los tipos de capacidades o dominios que aprende el sujeto:

- Destrezas motoras o habilidades del sistema muscular.
- Información verbal: nombres, hechos, generalizaciones.
- Habilidades intelectuales: discriminaciones y cadenas simples, conceptos, uso de reglas o principios, solución de problemas.
- Actitudes: creencias, valores, preferencias.
- Estrategias cognoscitivas: destrezas o habilidades organizadas internamente que gobiernan el propio aprendizaje del sujeto, su conducta, su pensamiento.

Chadwick (1975), citado, sugiere las relaciones entre los tipos de aprendizaje y los dominios de aprendizaje.

Las condiciones de aprendizaje en una situación de aprendizaje se identifican en términos de cuatro variables:

- Sujeto.
- Situación bajo la cual se hará el aprendizaje.
- Conducta de entrada: conocimientos previos.
- Conducta final que se espera del sujeto.

1.1.3 TEORIAS SOCIOCULTURALES.

Estas teorías plantean el aprendizaje como formas de socialización y enculturación. Subrayan la interacción con mediación entre el individuo y el medio ambiente, como se esquematiza en la figura 1.5, destacando la interacción individuo – grupo e individuo – medio ambiente (contexto). Su principal manifestación es el aprendizaje contextual compartido.

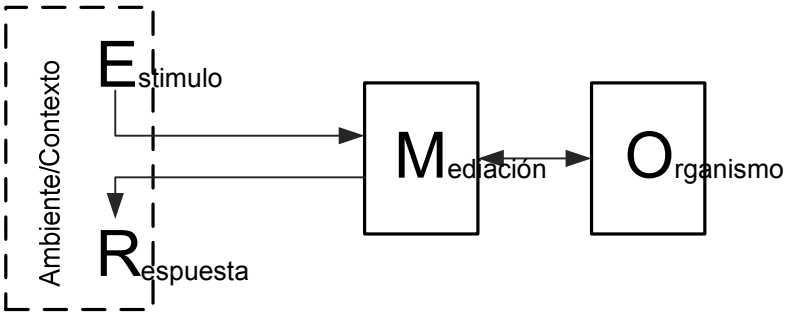


Figura 1.5 Modelo de aprendizaje E-M-O-M-R

Lev Vygotsky (1926), citado, con su teoría "Zona de desarrollo próximo", concibe al hombre como una construcción más social que biológica con funciones superiores logradas como fruto mediado del desarrollo cultural, donde el motor del aprendizaje es la actividad del sujeto condicionada por dos tipos de mediadores: "herramientas" y "símbolos", ya sea autónomamente en la "zona de desarrollo real", o ayudado por la mediación de otros en la "zona de desarrollo potencial". Las herramientas son las expectativas y conocimientos previos que trasforman los estímulos informativos que llegan del contexto. Los símbolos son signos que utiliza el sujeto para hacer propios dichos estímulos.

Las herramientas y símbolos son artificiales (naturaleza social); de modo que el aprendizaje del sujeto reside en su incorporación a la cultura, en el sentido del aprendizaje del uso de los sistemas de signos o símbolos que los hombres han elaborado a lo largo de la historia, especialmente el lenguaje (función psicológica superior). Este proceso se denomina "ley de la doble formación": el conocimiento se adquiere procesándolo, primero, desde el exterior, con las "herramientas" y reestructurándolo luego en el interior, a través de los "símbolos".

La importancia del ambiente como escenario de conducta y del desarrollo humano es tratada por varios autores en distintas épocas, entre los que se destaca Urie Bronfenbrenner (1987), citado, quien plantea que el ambiente está compuesto por las estructuras concéntricas: microsistema, mesosistema, exosistema y macrosistema.

El microsistema corresponde a escenarios inmediatos como hogar, escuela, grupo, con su patrón de actividades, papeles y relaciones interpersonales.

El mesosistema es un sistema que incluye escenarios inmediatos en los cuales participa la persona activamente. Ejemplo: hogar y grupo de amigos,

El exosistema corresponde a escenarios externos que afectan indirectamente el desarrollo de la persona. Ejemplo: lugar de trabajo del padre.

Macrosistema referido a la consistencia en forma y contenido de los sistemas anteriores dentro de una determinada cultura/subcultura.

Un quinto sistema fue adicionado con el nombre de cronosistema, relacionado con un orden de eventos ambientales y transiciones a los largo de la vida.

1.2 ELEMENTOS DEL PROCESO DE ENSEÑANZA – APRENDIZAJE.

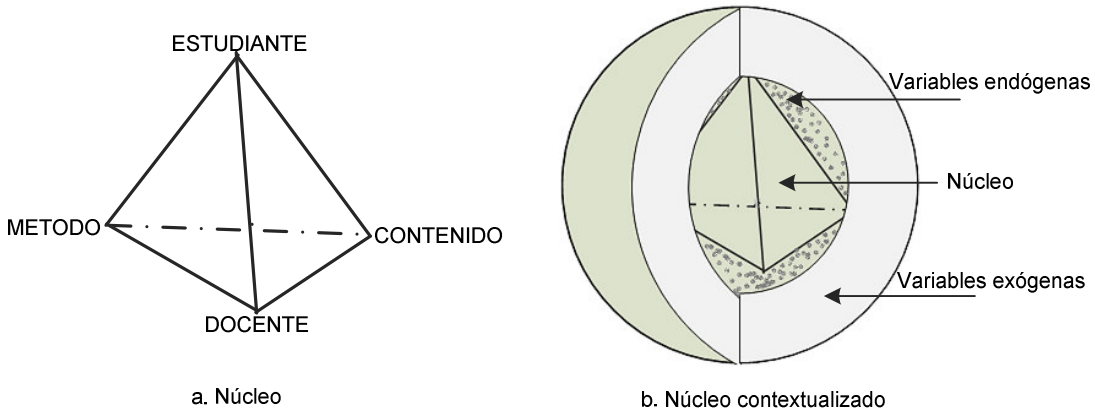
Las formas como ocurre el aprendizaje, sustentadas por las teorías vigentes, condicionan diferentes modalidades de enseñanza orientadas a lograrlo, que vinculan los procesos de enseñar y los procesos de aprender en diferentes versiones del proceso de enseñanza – aprendizaje (PEA). Contreras (1990), citado en [11], plantea al PEA como un sistema de comunicación intencional que se produce en un contexto y en el que se dan estrategias para provocar el aprendizaje.

Diferentes autores presentan al docente, al estudiante, al contenido, al contexto…, como elementos implicados en el complejo PEA, que toma forma según el elemento que protagoniza, los elementos que participan... Entre las formas propuestas se distinguen: facilitador del aprendizaje, relación comunicativa, comunicación mediada, sistémico – estructural, generador de interacción… En adelante se presenta esta última forma, propuesta por Adalberto Ferrández en la década de los 90, citado por [11].

El PEA se plantea como la interacción intencional y sistemática del docente y del estudiante en situaciones probabilísticas usando las estrategias más propias para integrar los contenidos culturales, poniendo en actividad todas las capacidades de la persona y pensando en la transformación socio-cultural del contexto que le es patrimonial. La interacción de los elementos básicos (núcleo): docente, estudiante, método y contenido, se modifica por el contexto (quinto elemento) espacial y temporal; resultando en un PEA flexible. El modelo de este PEA, cuyos elementos se resumen en los siguientes párrafos, se presenta en la figura 1.6, donde se destacan:

- El núcleo de cuatro elementos, mostrado en la figura 1.6.a. totalmente relacionados: docente (profesor, formador), estudiante (alumno, discente), método (estrategias metodológicas) y contenido cultural que se tiene que integrar significativamente.

- El contexto espacial y temporal, mostrado en la figura 1.6.b., compuesto por variables endógenas y exógenas. Las primeras son consecuencias de la realidad escolar concreta: recursos humanos, materiales y funcionales. Las segundas provienen del entorno social en el que se produce el aprendizaje: la política educativa, el ambiente cultural, el sistema de creencias, etc.



a. Núcleo

b. Núcleo contextualizado

Figura 1.6 Modelo del proceso enseñanza – aprendizaje de Ferrández

1.2.1 CONTEXTO.

El PEA cuenta con características específicas desde la educación obligatoria en un centro educativo hasta la enseñanza que se desarrolla en un centro de ocio o en el medio ambiente físico y social específico; dando lugar a la enseñanza formal, no formal e informal.

Los elementos del contexto identificados como presencialidad determinan los modelos de formación: presencial, semipresencial, a distancia, virtual.

La organización de elementos del contexto por las instituciones que generan la enseñanza – aprendizaje da lugar a modalidades organizativas de formación: escolar, de mercado, mediacional, virtual.

Los **medios y recursos** del entorno, siempre y cuando sean bien utilizados, actúan como facilitadores que mejoran el PEA.

1.2.2 DOCENTE.

Realiza los procesos de enseñar, actuando como formador, guía, facilitador, mediador, experto, estimulador, ayuda del estudiante…, según sus competencias, como persona, como agente activo..., en el entorno del PEA.

El docente se caracteriza por su actitud innovadora y el dominio de competencias profesionales. Innova buscando nuevas posibilidades de enseñanza – aprendizaje, empleando las potencialidades de los medios y recursos. Las competencias profesionales incluyen: saber sobre el tema, saber hacer, saber ser y hacer saber. Estas características permiten evitar situaciones de falta de uso de aportaciones como enseñanza programada, enseñanza asistida por computador…

Particularmente en la enseñanza formal y por el momento de su aplicación, las competencias del docente se agrupan en: preactivas (planificación del PEA), interactivas (desarrollo del PEA) y postactivas (evaluación del PEA).

Las competencias del docente pueden clasificarse en los siguientes dominios:

- Psicopedagógico: psicología del aprendizaje, estrategias metodológicas…
- Contenido del aprendizaje: integración, significatividad…
- Conocimiento sociocultural y sociolaboral.
- Medios y recursos: conocimientos generales y conocimientos sobre la aplicación en el PEA.

1.2.3 ESTUDIANTE.

Realiza los procesos de aprender dentro del PEA a partir de las indicaciones del docente, mediante la interacción con los medios y recursos del contexto.

El estudiante media en su proceso de aprendizaje vía el procesamiento de información que realiza, influido por el procesamiento que el docente hace de esa información.

El procesamiento de información que realiza el estudiante (almacenamiento, proceso, recuperación y uso) se apoya en los medios. En consecuencia la naturaleza de estos medios influye en este procesamiento.

1.2.4 ESTRATEGIAS METODOLOGICAS.

Concebir al estudiante como un agente activo, realizar las actividades de aprendizaje, conseguir la motivación... son factores que exigen la implantación de estrategias metodológicas y la acción del profesor como mediador. Los elementos implicados: docente, estudiante, grupo, comunicación, medios y recursos, organización espacial y temporal... pueden relacionarse de maneras diferentes.

La variedad y flexibilidad de las estrategias metodológicas permiten una mayor riqueza perceptiva, una mayor motivación y una adecuación mayor a las diferencias individuales.

1.2.5 CONTENIDO DEL APRENDIZAJE.

El contenido constituye la base sobre la que actúan e interactúan las capacidades humanas en desarrollo: construcción del pensamiento, afectividad, ámbito psicomotor…

El contenido requiere relevancia, actualidad, objetividad y pertinencia; así como un carácter integrador que permitirá un aprendizaje estructurado y con significado.

El carácter integrador del contenido se da por los tipos de contenidos y su secuenciación en el PEA. Los tipos de contenidos son: conceptuales, procedimentales y actitudinales.

Los contenidos conceptuales se refieren a datos (contenidos factuales), conceptos y principios. Los contenidos factuales son hechos, acontecimientos, situaciones, datos, fenómenos… Los conceptos son objetos o símbolos con características comunes (perro, mamífero, número). Los principios son cambios de hechos, objetos o situaciones en relaciones con otros (leyes de la termodinámica, leyes del movimiento)

Los contenidos procedimentales incluyen técnicas y estrategias para adquirir y/o mejorar habilidades. Las técnicas son cadena de acciones organizadas. Las estrategias permiten planificar, tomar decisiones y controlar la aplicación de las técnicas.

Los contenidos actitudinales se refieren a actitudes y valores. Las actitudes son disposiciones afectivas y racionales que se manifiestan en el comportamiento por medio de tendencias o disposiciones adquiridas a evaluar un objeto, persona, suceso, situación…, y a actuar consecuentemente. Los valores forman escala de valores, conjunto de principios, que permiten emitir juicios de valor.

En el siguiente capítulo se presenta el hardware y el software del computador.

Capitulo II:

EL COMPUTADOR

Dos tipos de componentes esenciales, integrados y complementados, constituyen el computador: el hardware y el software. Como se muestra en la Figura 2.1, en la base se ubica el hardware compuesto por los diferentes componentes físicos del computador. Sobre el hardware se implanta software; el cuál tiene como base al sistema operativo.

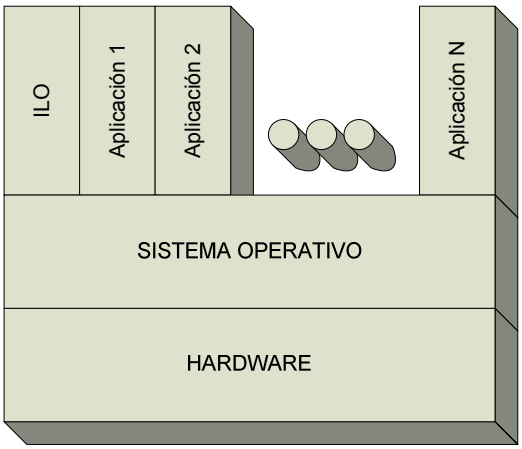


Figura 2.1 Computador o sistema informático

Este capítulo empieza describiendo la arquitectura Von Newman del hardware del computador, tema extensamente desarrollado desde múltiples perspectivas por muchos autores como [4], [10], [13], [15]...; y termina presentando los aspectos importantes del sistema operativo, tema igualmente tratado desde diferentes perspectivas por muchos autores, incluyendo [5], [12], [14], [16], [17]…

2.1 ARQUITECTURA HARDWARE DEL COMPUTADOR.

El computador digital de programa almacenado definido en el modelo propuesto por von Neumann está compuesto de las unidades debidamente interconectadas siguientes: unidad de entrada (UE), unidad proceso (UP), unidad de salida (US), unidad de memoria (UM) y unidad de control (UC). La figura 2.2 presenta esta versión de modelo, donde la información que entra del mundo exterior por la UE es procesada en la UP y los resultados entregados al mundo exterior vía US o almacenados en la UM para posterior tratamiento. Este proceso es controlado por un programa previamente

almacenado en la UM, desde donde la UC lo lee, lo interpreta y genera las respectivas direcciones y señales de control para cada una de las unidades. En la figura las líneas negras representan las rutas de direcciones y de señales de control.

Estas unidades funcionales deben ser alimentadas y alojadas en condiciones apropiadas, requerimientos realizados por la fuente de poder y los dispositivos de alojamiento, que no figuran en el modelo.

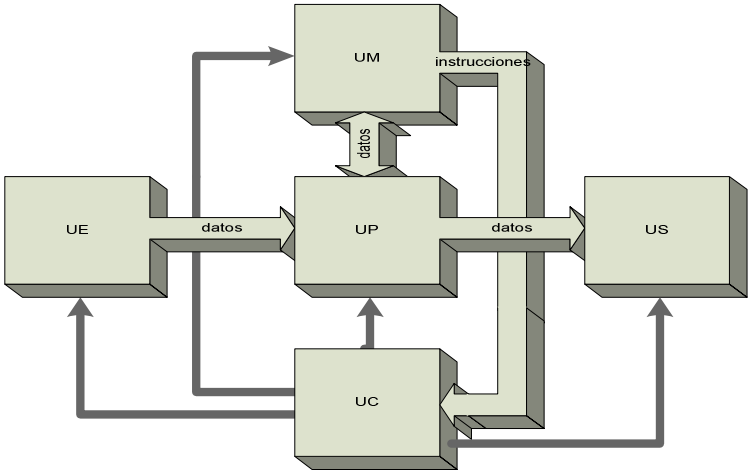


Figura 2.2: Arquitectura del computador "modelo de Von Newman"

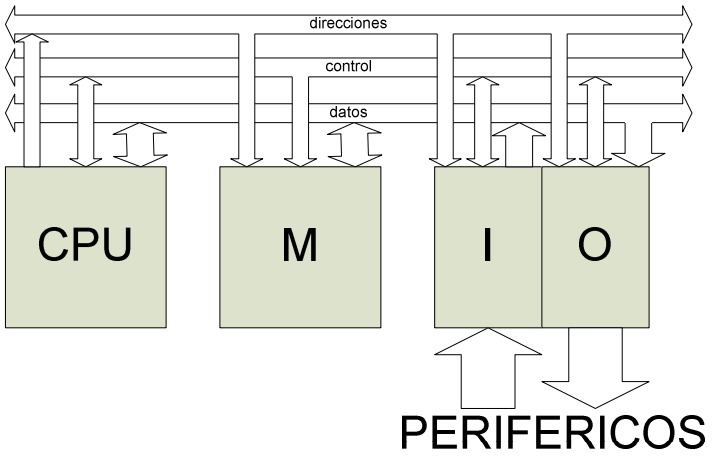


Figura 2.3: Arquitectura del computador con buses

Con el devenir de la tecnología, la organización e interconexión del hardware del computador según el modelo de von Neumann ha evolucionado a la forma presentada en la figura 2.3. Las funciones de la UP y de la UC se han integrado en la unidad central de proceso (CPU) o simplemente procesador (P) que implementado con tecnologías microelectrónicas constituye el microprocesador (µP). La UE se ha organizado en interfaz (control) de entrada y periférico de entrada, al igual que la US – en interfaz (control) de salida y periférico de salida - constituyendo la interfaz de entrada y salida (I/O) y los dispositivos periféricos. La interconexión entre los

subsistemas resultantes se ha sistematizado a través de un sistema de conexión con alto nivel de estandarización conocido como buses del sistema (computador). Según el tipo de información que transportan, se distinguen bus de datos, bus de direcciones y bus de control.

2.1.1 PROCESADOR.

El procesador es el "cerebro" del computador, decide la instrucción en lenguaje máquina a ejecutar, la decodifica y ejecuta. Es el diseño de la CPU lo que distingue al computador de programa almacenado de los demás sistemas digitales electrónicos. El procesador está compuesto de la unidad de proceso y la unidad de control tal como se muestra en la figura 2.4.

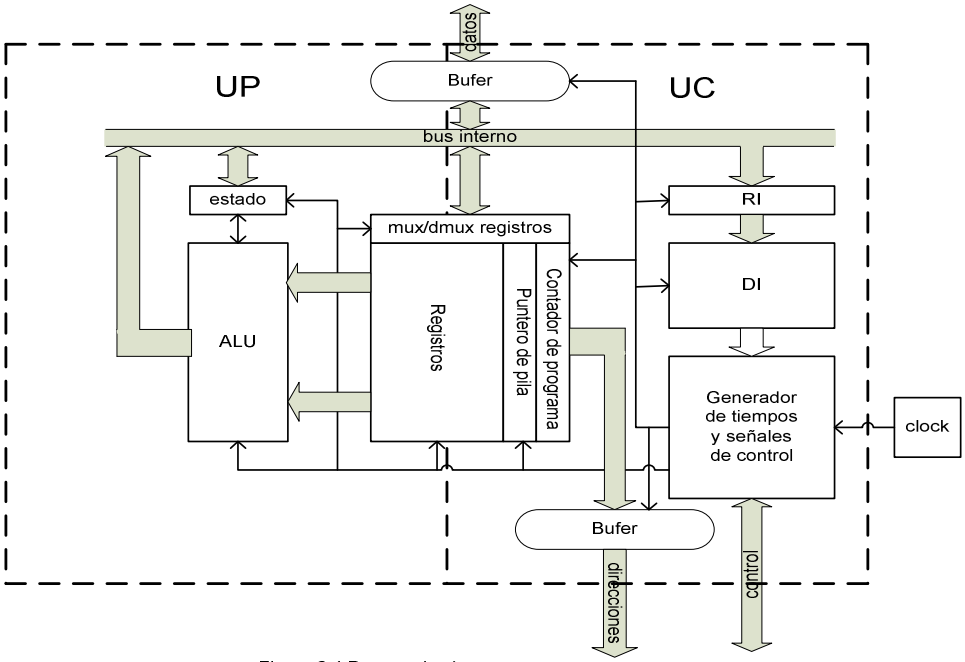


Figura 2.4 Procesador (μP)

2.1.1.1 Unidad de proceso.

En la unidad de proceso (UP) se realizan diferentes tipos de operaciones:

- aritméticas: suma, resta, multiplicación, división
- lógicas: AND, OR, XOR NOT
- transferencia entre registros.
- Relativas a registros e indicadores de estado

Para llevar a cabo las operaciones, la UP cuenta con la unidad funcional o unidad aritmética - lógica (ALU), registros de propósito general y registros de estado o

de control debidamente interconectados por el bus interno y sincronizados por las señales de control generadas por la UC..

Las operaciones se realizan sobre un operando (operación unaria) o sobre dos operandos (operación binaria). Los operandos son representaciones de datos en determinados códigos binarios, que emplean solamente dos dígitos: 0 y 1. Los operandos se toman de registros, memoria primaria o interfaz I y el resultado de la operación se dirige a un registro, memoria primaria o interfaz O. Sin embargo es frecuente realizar transferencia entre memoria y registros y realizar operaciones entre registros y ALU

Las operaciones suelen afectar a los indicadores de estado (incluidos en los registros de estado). Si se restan dos números iguales, el indicador de cero pasa a valer 1, indicando que el resultado ha sido cero. Si se resta un número mayor de un número menor, el indicador de signo pasa a valer 1, indicando que el resultado es negativo. Entre los indicadores más comunes que se implementan en los procesadores se encuentran: modo de operación, cero, signo, paridad, acarreo, auxiliar, dirección, interrupción, desborde, desvío, parado. Los indicadores de estado condicionan la ejecución de determinadas instrucciones del procesador como los saltos condicionales o llamadas a subrutinas condicionales.

Para ejecutar el cálculo expresado en el siguiente fragmento de código fuente, de nivel de un lenguaje de programación general:

```
a = b - c;
d = a + 25;
```

el procesador, por ejemplo, deberá realizar las operaciones correspondientes a las instrucciones en lenguaje máquina (expresadas en lenguaje ensamblador) que se dan a continuación:

```
//instrucciones en lenguaje ensamblador para a = b - c
carga R1, b       //transferencia de memoria primaria a registro de propósito general
carga R2, c       //transferencia de memoria primaria a registro de propósito general
resta R1, R2      //ALU resta el valor de R2 del valor de R1 y dirige resultado a R1
almacena R1,a     //transferencia de registro de propósito general a memoria primaria
//instrucciones en lenguaje ensamblador para d = a + 25
cargaD R2,25      //transferencia de memoria primaria a registro de propósito general
```

suma R1, R2 /ALU suma el valor de R2 al valor de R1 y dirige resultado a R1
almacena R1, d //transferencia de registro de propósito general a memoria primaria

La ALU es el motor de cómputo real del computador. La unidad funcional puede ser simple o compleja. Las operaciones de punto flotante son más complejas que las operaciones sobre enteros, razón por la cual algunas unidades funcionales no incorporan la aritmética de punto flotante. En estos casos las operaciones de punto flotante se realizan por software o en un procesador auxiliar dedicado.

2.1.1.2 Unidad de control.

La unidad de control (UC) gestiona la obtención (fetch) y la ejecución de las instrucciones en lenguaje máquina que constituyen los programas ejecutables y que se encuentran almacenados en la memoria primaria. Para cumplir con esta gestión, como se aprecia en la figura 2.4, la UC cuenta con un registro de instrucciones (IR), un decodificador de instrucciones (ID), un contador de programa (PC), un puntero de pila (SP), registros auxiliares (AR) , un generador de tiempos y señales de control (BTS) y comparte los indicadores con la UP,

El PC contiene la dirección de la instrucción durante la obtención y la dirección de la siguiente instrucción durante la ejecución. Si la instrucción ocupara varias direcciones de memoria (está formada por varias palabras), el PC se incrementa varias veces para obtener toda la instrucción. El contenido inicial del PC lo determina el fabricante del procesador. El PC está íntimamente ligado con la pila (stack), con la cual intercambia contenidos como las direcciones para el retorno de subrutinas y de interrupciones. El stack puede implementarse en el procesador o en memoria.

El SP permite direccionar cual de los registros del stack esta activo o latente. Si el stack se encuentra en memoria el contendido del SP sale por el bus de direcciones para indicar una dirección de memoria.

La gestión de direcciones de memoria en general involucra el PC, el SP, AR y otros componentes, como la aritmética de direcciones, para implementar diferentes mecanismos de direccionamiento, entre los cuales se encuentran la indexación, el desplazamiento.

El IR captura la instrucción obtenida de la memoria durante el fetch y lo mantiene durante su ejecución. La parte del código de la instrucción lo traspasa el ID para decidir que debe hacerse posteriormente con el concurso del BTS.

El BTS incorpora la base de tiempos y el secuenciador o generador de microinstrucciones que en conjunto generan las señales de temporización y control requeridas para ejecutar las diferentes fases u operaciones por cada una de las unidades de un computador digital.

El tiempo elemental de la base de tiempos y de funcionamiento del procesador, que determina la velocidad de ejecución de instrucciones (velocidad de proceso), está definido por el clock (reloj) externo dentro del margen que soporta el procesador. La frecuencia (velocidad) del reloj del procesador alcanza hasta gigahercios. En consecuencia, las instrucciones se ejecutan en microsegundos, inclusive en nanosegundos; velocidades imperceptibles a la vista humana.

2.1.2 MEMORIA.

La unidad de memoria o memoria primaria del computador almacena los programas y datos mientras están siendo manipulados por el procesador. La memoria, según se ve en la figura 2.5, está compuesta de una interfaz con los buses del sistema y el almacén.

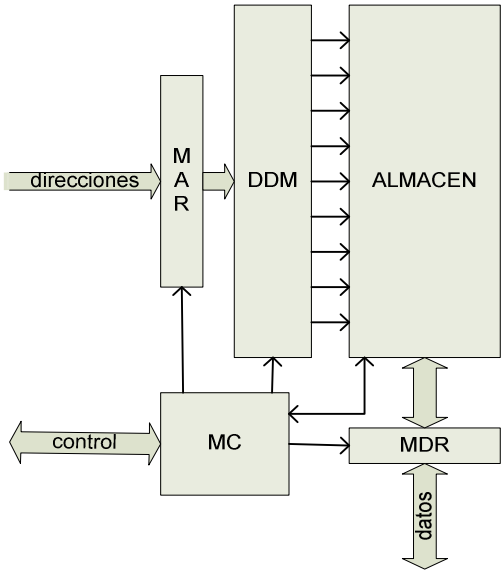


Figura 2.5 Memoria primaria RAM

La interfaz con los buses del sistema incluye el registro de direcciones de memoria (MAR), el decodificador de direcciones (DDM), el registro de datos de memoria (MDR) y el bloque de control de memoria (MC)

El MAR captura la dirección desde el bus de direcciones y lo mantiene durante el tiempo requerido para la escritura o lectura de la memoria. Esta dirección es decodificada por el DDM para seleccionar una palabra del almacén.

El almacén está compuesto por grupos de biestables denominados palabras de memoria, cada biestable almacena un bit. Las palabras de memoria son direccionables, no sus elementos (bits).

En la escritura el MDR captura, del bus de datos, el dato (palabra de datos) a escribirse en el almacén. En la lectura el MDR captura el dato entregado por el almacén y lo hace disponible en el bus de datos del sistema.

Para escribir en la memoria, el dato se coloca en el MDR, la dirección de memoria se coloca en el MAR y se suministra al MC la orden de escritura. Tras un ciclo de escritura, la UM guarda el contenido del MDR en la palabra del almacén direccionada por el MAR.

Para leer la memoria, se coloca una dirección en el MAR y se suministra la orden de lectura al MC. Tras un ciclo de lectura, la UM copia el contenido de la palabra del almacén, direccionada por el MAR, en el MDR. El dato del MDR está disponible en el bus de datos, desde donde su puede capturar por otra unidad.

El número de palabras de la UM y la anchura de ellas están determinados por la tecnología de fabricación y por las consideraciones de diseño del hardware.

2.1.3 PERIFERICOS E INTERFAZ E/S.

Existe una gran variedad de tipos de dispositivos periféricos que se conectan al computador por medio de interfaces de entrada/salida (E/S), un caso particular se muestra en la figura 2.6.

Un dispositivo de entrada, como un teclado, un ratón, una pantalla táctil o un micrófono, transfiere datos al procesador y éste puede almacenarlos en memoria. El procesador obtiene datos de memoria y los transfiere a un dispositivo de salida, como una pantalla, un parlante o una impresora. Los dispositivos de comunicaciones, como

modems, receptores/trasmisores cableados e inalámbricos, son dispositivos de entrada y salida. Los dispositivos de almacenamiento, como discos magnéticos, discos ópticos o cintas, también son dispositivos de entrada y salida.

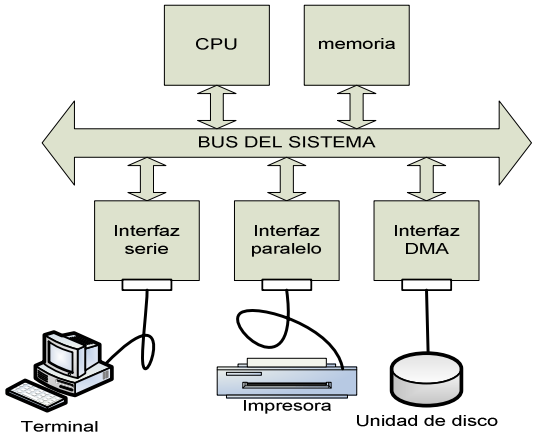


Figura 2.6 Computador con periféricos

La mayoría de dispositivos periféricos operan asincrónicamente respecto al procesador y a velocidades que van desde unas pocas transferencias hasta millones de transferencias de datos por segundo.

La operación asincrónica exige que el hardware de E/S incluya señales de diálogo (handshaking) que capaciten al procesador e interfaz de E/S (controlador de dispositivo) para que se indiquen mutuamente sus estados, intenciones y disponibilidades de datos. Cuando el procesador requiere un dato de entrada consulta el estado del dispositivo. Si el dato está disponible el procesador lo obtiene de la interfaz, en caso contrario el procesador tiene que esperar o dedicarse a otra tarea (multitarea) hasta que termine la operación de entrada desde el dispositivo.

Los dispositivos periféricos exhiben velocidades de trasferencia desde el orden de menos de 1 cps (caracteres por segundo) hasta el orden de millones de cps. El extremo inferior del espectro lo representan los terminales lentos, conmutadores y visualizadores; el extremo superior pertenece a los enlaces de comunicación óptica y unidades de disco de alto rendimiento. Los dispositivos de muy alta velocidad suelen conectarse al computador mediante canales de E/S especializados o procesadores dedicados, utilizando técnicas de acceso directo a memoria (DMA). Los dispositivos menos exigentes usan E/S programada, donde cada transferencia es controlada por el procesador.

La mejora de la utilización del procesador implica la asistencia de hardware de E/S en la forma de búferes de datos, señales de diálogo y mecanismos de sincronización tales como las interrupciones, aspectos que se presentan a continuación.

2.1.3.1 Interfaces de entrada/salida.

Los controladores hardware o interfaces de E/S median entre el computador y los dispositivos de E/S, tal como se esquematiza en la figura 2.6, para superar las incompatibilidades de velocidad y de señalización entre el procesador y los periféricos y traducir las órdenes de E/S genéricas emitidas por el procesador a controles específicos del dispositivo.

Como indica la figura 2.6, los controladores de E/S están encargados de los dispositivos serie, paralelo y DMA. Muchos terminales y líneas de comunicación se conectan por medio de interfaz serie al computador. La mayoría de impresoras, conmutadores, relés, leds y otros visualizadores orientados a líneas se conectan por medio de interfaz paralela al computador. La interfaz de tipo DMA es mas utilizada por dispositivos de alto rendimiento como discos, cintas, redes de área local y multiplexores de terminales sofisticados.

Un controlador se diseña para controlar uno o más dispositivos del mismo tipo o de tipos muy similares. Los dispositivos múltiples, cuando se soportan, son gestionados por medio de canales dedicados (uno por cada dispositivo) o en modo cadena de eslabones (daisy-chain). La primera modalidad es común en controladores de terminal, mientras que la segunda se encuentra en controladores de disco.

La estructura genérica de un controlador de E/S se presenta en la figura 2.7 y consta de tres capas funcionales: interfaz al bus del sistema, controlador genérico e interfaz al dispositivo.

La interfaz al bus incluye la lógica de direcciones y control, así como los conductores al bus de datos, a fin de que el controlador aparezca al bus del sistema como un módulo estándar, capaz de participar en las transacciones, tales como lecturas, escrituras, arbitrajes de prioridad, interrupciones y ciclos de DMA.

La interfaz al dispositivo incluye parte del controlador específico del dispositivo y el acondicionamiento de señales a los niveles eléctricos requeridos para ser compatible con el dispositivo periférico. Estas características son dependientes del

dispositivo y típicamente bastante diferentes de los niveles lógicos estándares del computador. Los componentes electromecánicos que se encuentran en muchos periféricos requieren emplear una variedad de niveles de tensión y corriente digitales y con frecuencia algunas señales analógicas. Esta situación restringe a que cada controlador sirva como interfaz de solo una colección específica de dispositivos físicos funcionalmente similares.

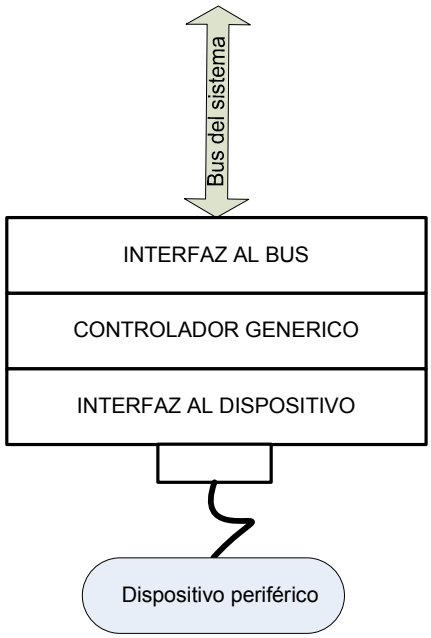


Figura 2.7 Estructura genérica de la interfaz

El controlador genérico proporciona una abstracción uniforme de los dispositivos de E/S compuesta por un conjunto de registros dedicados: registros de datos, registros de estados y registros de órdenes. Las consultas de estados, la emisión de órdenes de E/S específicas de dispositivo y las transferencias de datos se consiguen leyendo o escribiendo en esos registros. A un conjunto de tales registros dedicados se les denomina generalmente puerto de E/S. Las variedades más habituales de puerto de E/S incluyen puertos paralelos, puertos serie, controladores de disco duro y disquete, controladores gráficos, temporizadores de intervalo programable, controladores de interrupción programables, controladores de LAN, USARTs. Las tecnologías LSI y VLSI permiten poner varios puertos de E/S en un circuito integrado.

En la figura 2.8 se muestra la estructura de un puerto de E/S típico, conectado al bus del sistema y al dispositivo de E/S en sus respectivos extremos y conteniendo el control genérico, compuesto por los siguientes tipos de registros: registros de datos (búferes de entrada y salida), registro de estado y registro de órdenes.

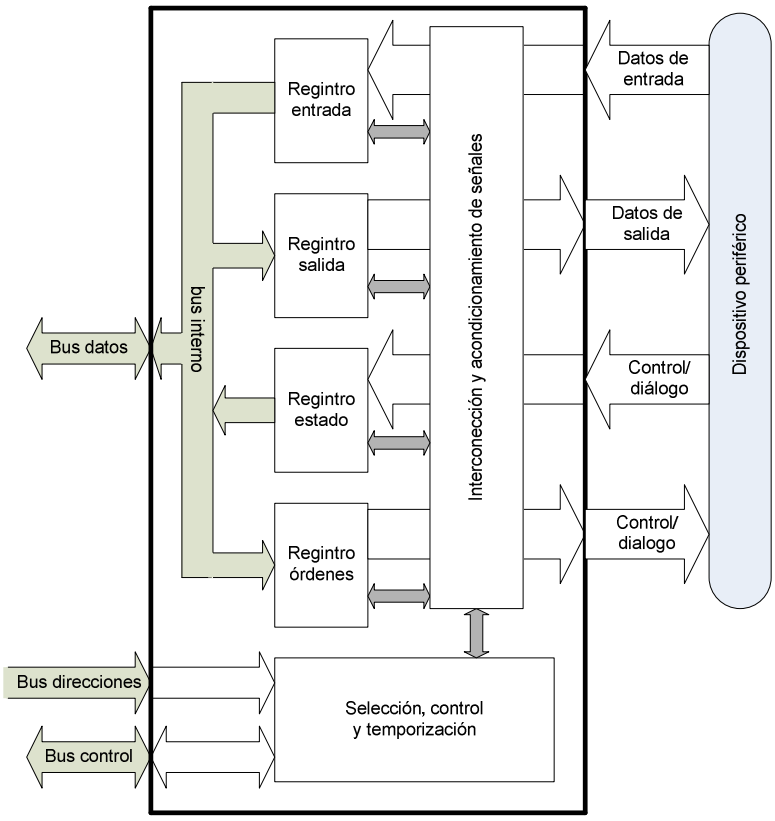


Figura 2.8 Puerto genérico de E/S

Los registros (búferes) de entrada y salida se utilizan para contener los datos en tránsito hasta que el dispositivo de E/S y el procesador estén preparados y en disposición de concluir la transacción de E/S. Los registros de órdenes y de estado son parte del mecanismo de diálogo para implementar el protocolo de negociación y sincronizar momentáneamente el procesador con los dispositivos de E/S a fin de intercambiar datos.

El registro de órdenes está encargado de transferir órdenes de E/S del procesador al dispositivo de E/S. Las órdenes de E/S tienden a agruparse en dos categorías generales: de modo y operacionales. Las órdenes de modo seleccionan un modo particular de operación (entrada o salida), un algoritmo de chequeo de errores o un protocolo de comunicaciones. Las órdenes operacionales de E/S gobiernan la temporización y el mecanismo de transferencia de datos. Las primeras se emiten una vez, durante la inicialización del dispositivo, las últimas se emiten tantas veces como sea necesario, típicamente una vez por cada transferencia de datos.

El registro de estado se utiliza para proporcionar información al procesador sobre el estado del dispositivo de E/S, tal como dispositivo listo u ocupado, búfer lleno

o vacío e indicaciones de error. Los registros de estado se implementan como una colección de bits de solo lectura, cada uno asociado con alguna condición. La operación habitual es comprobar el valor de un bit para determinar el estado de la condición que representa.

Las direcciones de los puertos pueden estar incluidas en el espacio de direcciones de la memoria (puertos mapeados en memoria) o formar su propio espacio (E/S aislada). Los puertos mapeados en memoria pueden ser accedidos y manipulados utilizando el repertorio completo de instrucciones de referencia a memoria. Los puertos no mapeados en memoria se acceden y manipulan por medio de instrucciones especiales, tales como INPUT y OUTPUT, que activan las direcciones aisladas de E/S.

2.1.3.2 Entrada/salida controlada por programa.

En la figura 2.9 se presenta una transacción de entrada controlada por programa que involucra un dispositivo y el procesador. La transacción de salida consta de los mismos pasos en un orden ligeramente cambiado.

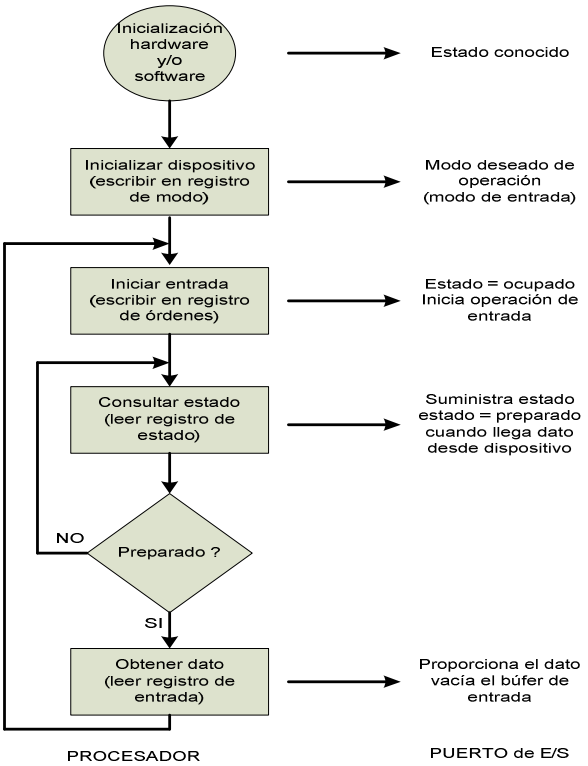


Figura 2.9 Entrada controlada por programa

El arranque o reinicialización del computador pone el dispositivo en un estado conocido por medio de inicialización hardware y/o software. La inicialización del

dispositivo selecciona el modo deseado de operación (modo entrada). Eventualmente puede cambiarse de modo, por ejemplo a modo salida.

Antes de cada transacción de entrada, el procesador indica su deseo de recibir datos por medio de una orden "iniciar entrada", escribiendo el patrón de bits correspondiente en el registro de órdenes del puerto de E/S de la interfaz respectiva. Como consecuencia, el puerto de E/S activa el bit ocupado, indicando que hay una operación en progreso. Cuando los datos están disponibles en el búfer de entrada, el puerto de E/S activa el bit preparado. El procesador puede determinar el valor de los bits de estado leyendo el registro de estado. Cuando descubre que el puerto está preparado, el procesador lee los datos del búfer de entrada. Después se inicia otra transacción de entrada y se repite el ciclo completo.

El bucle interno de la figura 2.9 se ejecuta hasta que la interfaz este preparada para transferir el dato. Este tipo de bucle utilizado tanto para entrada como para salida se denomina bucle de espera activa y es la base del método de E/S controlado por programa. Este método de E/S se caracteriza por baja utilización efectiva del procesador y dificultad para manejar múltiples dispositivos de E/S.

El bucle de espera activa puede ser ampliado para manejar múltiples dispositivos de E/S por medio de encuesta (polling), técnica que tiene la siguiente estructura:

Comprobar estado del dispositivo 1, si está preparado ir al servicio de E/S 1
Comprobar estado del dispositivo 2, si está preparado ir al servicio de E/S 2
. . .
Comprobar estado del dispositivo n, si está preparado ir al servicio de E/S n
Repetir la secuencia de encuesta.

La encuesta debe manejar la concurrencia de dispositivos preparados y sus prioridades

La utilización efectiva del procesador puede mejorarse realizando la encuesta periódicamente y ejecutando otra tarea (multitarea) en el resto del tiempo. La encuesta periódica agrega a los problemas anteriores el manejo de la periodicidad de la encuesta y sus consecuencias.

En el siguiente párrafo se plantea la solución a estos problemas introduciendo el mecanismo de interrupción.

2.1.3.3 Entrada/salida guiada por interrupciones.

La interrupción es un mecanismo asistido por hardware para sincronizar el procesador con los sucesos (asíncronos). Por medio de este mecanismo, como se esquematiza en la figura 2.10, la interfaz de E/S fuerza al procesador a abandonar temporalmente su actividad actual y prestar servicio a los sucesos de E/S cuando ocurran. Después de dar servicio a un dispositivo ejecutando su gestor de interrupción (GI) asociado, el procesador reanuda la actividad abandonada desde el punto de la interrupción. Entre los estados de interfaces que generan interrupción se encuentran los siguientes: dato de entrada preparado, disponibilidad para dato de salida y transferencia DMA finalizada.

El programa actual se ejecuta en el procesador (dentro de su contexto, formado por los valores de registros e indicadores) hasta que ocurre un suceso de E/S y genera una petición de interrupción. La petición de interrupción suele generar un ciclo de interrupción del procesador. Durante el ciclo de interrupción, el hardware del procesador guarda el PC y el estado (indicadores) del programa interrumpido en la pila, deshabilita el indicador de interrupción y carga en el PC la dirección del GI. Según el tipo de procesador, el ciclo de interrupción puede ejecutar otras operaciones elementales. El PC y el estado del programa interrumpido serán recuperados para su reanudación. Esta recuperación suele realizarse por medio de un retorno de interrupción (RTI) que suele ser la última instrucción del GI.

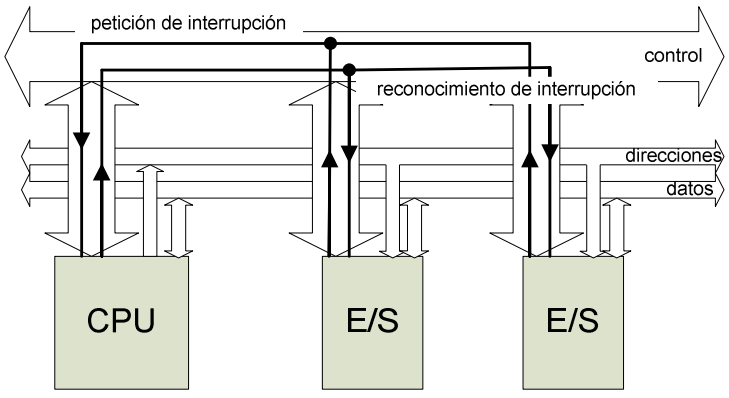


Figura 2.10: Señales de interrupciones

El GI salva el contexto (definido para salvar y no guardado por hardware), da servicio a la interrupción, notifica a la interfaz el reconocimiento de interrupción (para que ésta retire la señal de petición de interrupción), restaura el contexto que salvó y ejecuta el RTI (reanudándose el programa interrumpido).

Excepto porque salva y restaura contexto, un GI no es más que una secuencia de código que se ejecuta en respuesta a un suceso externo. Como tal, consta de instrucciones para transferir datos, verificar existencia de errores y actualizar punteros; estructura muy similar al servicio de E/S controlada por programa. No es necesario verificar la disponibilidad del dispositivo; ya que el GI se invoca por la señal de solicitud de interrupción (invocación hardware) cuando el dispositivo está disponible.

Determinados procesadores implementan una colección (vector) de niveles de interrupción debidamente priorizados por hardware. Cada nivel es servido por un GI, un elemento del vector de Gis, y puede recibir peticiones de interrupción de una o más interfaces. Las prioridades de las interfaces conectadas a un nivel pueden manejarse sobre la base de encuestas o por vectorización. Suele emplearse hardware auxiliar, controladores programables de interrupciones, para facilitar la gestión de la vectorización de interrupciones generadas por múltiples interfaces conectadas a un nivel de interrupción

2.1.4 REPERTORIO DE INSTRUCCIONES.

El hardware de un computador se caracteriza por su repertorio de instrucciones en lenguaje máquina con totalidad funcional. Un computador solo ejecuta programas expresados en instrucciones del lenguaje máquina de su repertorio. La totalidad funcional requiere de varios tipos básicos de instrucciones en lenguaje máquina:

- instrucciones aritméticas: suma, resta, multiplicación, división
- instrucciones lógicas: AND, OR, XOR NOT
- instrucciones relativas a registros: desplazamientos, circulaciones, incremento, decremento, complementaciones, borrado
- instrucciones relativas a indicadores: puesta a uno, puesta a cero
- instrucciones de intercambio de información entre registros de procesador
- instrucciones de intercambio de información entre registros y memoria
- instrucciones de carga inmediata de información en registros o memoria
- instrucciones de entradas/salida
- instrucciones de ruptura de secuencia: saltos in/condicionales, llamadas a rutinas in/condicionales, retorno de interrupción.
- instrucciones especiales y de control: parar, puesta en marcha, no operación.

Estas instrucciones se codifican en bits (código binario), constituyendo una o más palabras (bytes), y se realizan en uno o mas ciclos de máquina (memoria). Cada ciclo de máquina se implementa en unos cuantos ciclos o tiempos elementales definidos por el reloj (clock) del sistema conectado al procesador.

El computador desde que se activa (se prende), hasta que se desactiva (se apaga) y mientras el indicador parado esté deshabilitado, se encuentra ejecutando instrucciones de su repertorio. El indicador parar se establece con la instrucción parar y se restablece por medio de una interrupción. A esta realización continua de instrucciones a nivel hardware por parte del computador se suele denominar **proceso hardware** y tiene el siguiente esquema lógico.

```
PC <== <dirección inicial de máquina>
parado <== FALSE
mientras (no parado)
{
        IR <== memoria[PC]
        PC <== PC + 1;
        decodificar [IR]
        ejecutar[IR]
}
```

Un programa en lenguaje máquina (ejecutable) de un computador es una secuencia de instrucciones del repertorio del computador, que previamente se almacena en memoria primaria, desde donde es realizado (ejecutado) por el procesador a una velocidad definida por el reloj del sistema, en concordancia con el proceso hardware presentado.

2.2 SISTEMA OPERATIVO.

El sistema operativo (SO) actúa como una interfaz entre los usuarios del computador y su hardware; así como un coordinador de los recursos del computador.

Como interfaz, el SO es una colección organizada de extensiones software al hardware, consistente en estructuras de datos y código que hacen funcionar en forma integrada los diferentes componentes del computador y proporcionan la máquina abstracta con varios componentes autónomos abstractos para la ejecución de programas

(procesos e hilos) tal como se muestra en la figura 2.11.. Los programas usan las facilidades proporcionadas por el SO, a través de su interfaz, para obtener acceso a los recursos del computador, tales como memoria, archivos y otros. Los programas invocan los servicios del sistema operativo por medio de llamadas (al SO). Los usuarios interactúan con el sistema operativo por medio de órdenes o instrucciones (al SO) procesadas por el interpretador de lenguaje de órdenes (ILO). Los servicios proporcionados por un SO dependen de varios factores. Los servicios visibles al usuario de un SO están determinados principalmente por las necesidades y características del entorno objetivo al que debe soportar. Así, un SO para soportar el desarrollo de programas en un entorno interactivo puede tener un conjunto bastante diferente de llamadas y órdenes que un SO destinado para soporte de ejecución de aplicaciones en tiempo real, tal como el control de una lavadora doméstica.

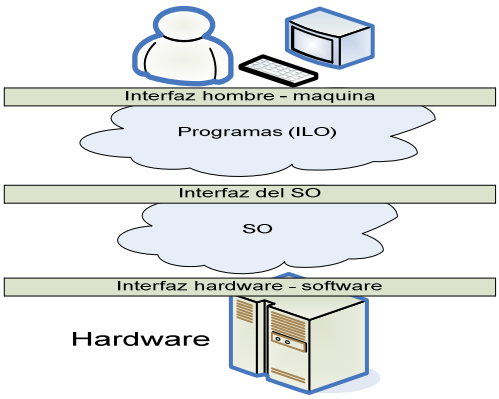


Figura 2.11 SO, programas y sus interfaces

El SO, como coordinador de recursos del computador, administra los recursos del computador tales como el procesador, la memoria, los archivos, los dispositivos y otros; compartiendo estos recursos vía multiproceso y multiprogramación entre múltiples maquinas abstractas tal como se muestra en la figura 2.12, donde se muestra la compartición de recursos entre tres máquinas abstractas. En esta función, el SO lleva la cuenta del estado de cada recurso y decide a que proceso (máquina abstracta) asignar un recurso, durante cuánto tiempo y cuándo. En sistemas que soportan ejecución concurrente de programas, el SO resuelve las peticiones conflictivas de recursos de manera que preserva la integridad del computador, y al hacerlo intenta optimizar el rendimiento.

En sistemas operativos de multiproceso o de multiprogramación un proceso (programa en ejecución) se realiza en una máquina abstracta.

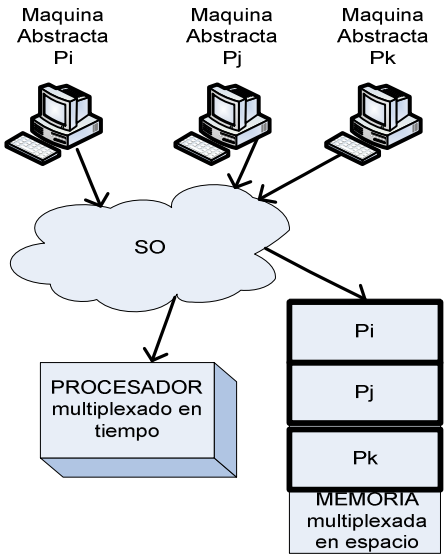


Figura 2.12 Multiprogramación

El objetivo principal de los sistemas operativos es incrementar la productividad de los recursos del computador y de los usuarios, objetivo generalmente contradictorio. Los sistemas operativos destinados a computadores (por ejemplo supercomputadores y determinados computadores especializados) donde la velocidad de proceso es un factor crítico se preocupan de obtener el máximo trabajo posible del computador; la conveniencia de los usuarios y su productividad pueden ser consideradas secundarias. Por otro lado, un SO de un computador personal se preocupa de la conveniencia y la productividad del usuario, dejando en otro orden utilización de los recursos de computador.

Para apreciar los aspectos básicos de los sistemas operativos, a continuación se presentan los principales tipos de sistemas operativos y sus características, los servicios genéricos que brindan desde los puntos de vista del usuario del lenguaje de órdenes y del usuario de las llamadas al sistema, sus funciones básicas y su arquitectura.

2.2.1 TIPOS.

Se consideran los siguientes tipos de sistemas operativos: de serie simple, de lotes, de multiprogramación y distribuidos. Las propiedades de cada tipo de SO se describen considerando los siguientes aspectos: gestión del procesador, gestión de la memoria, gestión de entrada/salida y gestión de archivos.

2.2.1.1 Sistemas operativos serie.

En el procesamiento en serie simple la ejecución completa de cada programa monopoliza el computador y suele requerir las siguientes actividades: preparación manual, remisión manual, ejecución por el computador y obtención de resultados.

El procesador se dedica al programa en ejecución y la mayor parte del tiempo no tiene trabajo productivo. La memoria es esquematizada en una parte para el SO residente y otra para la ejecución del programa del usuario. La asignación y la desasignación de dispositivos de E/S, cuando puede haber solo un programa en ejecución, es trivial. El acceso a los archivos es en serie.

2.2.1.2 Sistemas operativos de lotes.

El procesamiento por lotes requiere generalmente que cada programa, sus datos y las órdenes al sistema sean remitidos juntos en forma de trabajo. Se preparan manualmente y se remiten al computador un lote de trabajos (programas). Los sistemas operativos por lotes permiten poca o ninguna interacción entre los usuarios y los programas en ejecución. El procesamiento por lotes tiene un mayor potencial de utilización de recursos que el procesamiento serie simple en sistemas informáticos que dan servicio a múltiples usuarios. Debido a las dilaciones en los tiempos de retorno y a la depuración fuera de línea, el procesamiento por lotes no es muy conveniente para desarrollo de programas.

Los programas que no requieren interacción y los programas que tienen largos tiempos de ejecución pueden estar bien servidos por un sistema operativo de lotes. Entre ellos encontramos los programas de nóminas, de predicción del tiempo, de análisis estadístico y de grandes cálculos científicos. El procesamiento serie combinado con archivos de lotes de órdenes suele encontrarse en computadores personales.

Los trabajos son típicamente procesados en orden de llegada, pudiéndose usar otras estrategias para obtener una distribución más equilibrada de los tiempos de retorno.

La memoria suele dividir en dos áreas. Una de ellas está permanentemente ocupada por la parte residente del sistema operativo, y la otra es utilizada para cargar

programas durante su ejecución. Cuando un programa termina, se carga un nuevo programa en la misma área de memoria.

La ejecución de un programa a la vez no requiere ninguna gestión de dispositivo crítica en el tiempo. Muchos sistemas operativos de lotes utilizan el sencillo método de E/S controlada por programa. La falta de competencia por parte de los dispositivos de E,/S hace que su asignación y su desasignación sea trivial.

El acceso a los archivos es en serie. Esta modalidad de acceso requiere poca protección y ningún control de concurrencia.

2.2.1.3 Sistemas operativos de multiprogramación.

Una instancia de un programa en ejecución es denominada proceso o tarea. Un SO multitarea soporta la ejecución concurrente de dos o más procesos activos, cada uno en su máquina abstracta. La multitarea se implementa manteniendo el código y los datos de varios procesos simultáneamente en memoria, y multiplexando el procesador y los demás recursos del computador entre ellos, como se presenta en la figura 2.12. La multitarea se viabiliza con soporte de hardware y software para la protección y compartición de los recursos del computador por los procesos residentes.

El SO de multiprogramación soporta multitarea y proporciona formas mas avanzadas de protección de memoria y de control de concurrencia cuando los procesos acceden a dispositivos de E/S y archivos compartidos. Los sistemas operativos de multiprogramación que soportan múltiples usuarios se les denominan sistemas multiusuario. Los sistemas operativos multiusuario proporcionan entornos de usuario individuales, requieren autenticación de usuario y autorización de uso de recurso y proporcionan contabilidad de uso de los recursos por usuario. Un SO de multiprogramación usa la operación multitarea como un mecanismo para gestionar los recursos del computador: el procesador, la memoria, los dispositivos de E/S y los archivos. La operación multitarea sin soporte multiusuario suele encontrarse en sistemas operativos de computadores personales y en sistemas operativos de tiempo real.

Los sistemas operativos multiacceso permiten acceso simultáneo o concurrente a un sistema informático desde dos o más terminales, como es el caso de los sistemas de reservas aéreas, que soportan centenares de terminales activos bajo control de un único proceso.

Los sistemas operativos de multiprocesadores gestionan la operación de sistemas informáticos que incorporan varios procesadores. Los sistemas operativos multiprocesadores soportan la ejecución simultánea de múltiples procesos sobre diferentes procesadores. Dependiendo de la implementación, la multitarea puede o no estar permitida en los procesadores individuales.

En general, los sistemas de multiprogramación se caracterizan por una multitud de procesos simultáneamente activos que compiten por los recursos del computador. Un SO de multiprogramación vigila el estado de todos los procesos activos y de todos los recursos del sistema. Cuando se producen cambios importantes de estado, o cuando es invocado explícitamente, el sistema operativo se activa para asignar recursos y proporcionar ciertos servicios de su repertorio. Entre los sistemas operativos de multiprogramación se destacan los de tiempo compartido, los de tiempo real y los que integran estas funcionalidades.

2.2.1.3.1 Sistemas operativos de tiempo compartido.

Estos sistemas son representantes de los sistemas de multiprogramación, multiusuario. Los entornos generales de desarrollo de programas, los sistemas de diseño auxiliado por computador y determinados sistemas de procesamiento de texto requieran sistemas operativos de tiempo compartido.

Los sistemas de tiempo deben proporcionar un buen tiempo de respuesta de terminal, dando la ilusión a cada usuario de disponer de una máquina para sí mismo e intentando lograr una compartición equitativa de los recursos comunes. Cuando el sistema está cargado, los usuarios con mayores exigencias de procesamiento deberían experimentar tiempos de espera más largos.

Los sistemas de tiempo compartido generalmente utilizan una planificación por reparto rotativo del tiempo. Según esta estrategia, los procesos se ejecutan con prioridad rotativa que se incrementa durante la espera y disminuye después de que han recibido servicio. Para evitar que los procesos monopolicen el procesador, un proceso es interrumpido por el sistema si se ejecuta por mas tiempo que el establecido y colocado al final de la cola de procesos en espera. Este modo de operación proporciona generalmente rápidos tiempos de respuesta a programas interactivos.

La gestión de memoria en sistemas de tiempo compartido proporciona aislamiento y protección a los procesos residentes. En oportunidades se utilizan algunas formas de compartición controlada para conservar memoria y para intercambiar datos entre los procesos. Los procesos en sistemas de tiempo compartido no tienen grandes necesidades de comunicarse unos con otros, por pertenecer a diferentes usuarios.

La gestión de E/S debe ser suficientemente sofisticada para tratar con múltiples usuarios y dispositivos. Sin embargo, por las bajas velocidades de los terminales y de los usuarios, el procesamiento temporal de las interrupciones de terminal no es crítico.

Para soportar un acceso concurrente, la gestión de archivos en un SO de tiempo compartido debe proporcionar protección y control de acceso, a fin de que los archivos sean compartidos entre ciertos usuarios o clases de usuarios.

2.2.1.3.2 Sistemas operativos de tiempo real.

Es objetivo de los sistemas de tiempo real proporcionar rápidos tiempos de respuesta a sucesos y satisfacer así los plazos de planificación. Es frecuente que un sistema de tiempo real procese ráfagas de miles de interrupciones por segundo sin perder un solo suceso y que confíe generalmente en algunas políticas y técnicas específicas para realizar su trabajo.

En sistemas de tiempo real comúnmente surgen procesos explícitos definidos y controlados por el programador. Básicamente, cada proceso se encarga del manejo de un único suceso externo. El proceso se activa por la ocurrencia del suceso asociado expresada por medio de una interrupción. La operación multitarea se obtiene planificando los procesos para ser ejecutados, en forma independiente unos de otros. Cada proceso tiene asignado un nivel de prioridad que se corresponde con la importancia relativa del suceso al que sirve. Normalmente el procesador es asignado al proceso de máxima prioridad entre todos aquellos que están preparados para ejecutarse. Los procesos de mayor prioridad generalmente expropian la ejecución de los procesos de prioridad inferior. Esta forma de planificación, llamada planificación expropiativa basada en prioridades, es utilizada por la mayoría de los sistemas de tiempo real.

La gestión de memoria es comparativamente menos exigente que en otros tipos de sistemas de multiprogramación; dado que muchos procesos residen permanentemente en memoria con el fin de lograr tiempos de respuesta rápidos. La

población de procesos en sistemas de tiempo real es en gran medida estática, y hay comparativamente poco movimiento de programas entre a1macenamiento primario y secundario. Además, los procesos en sistemas de tiempo real tienden a cooperar estrechamente, necesitándose por tanto soporte para separación y compartición de memoria.

La gestión de dispositivos críticos en tiempo es una de las características principales de los sistemas de tiempo real. Además de proporcionar formas sofisticadas de gestión de interrupciones y de almacenamiento de E/S, los sistemas operativos de tiempo real suelen proporcionar llamadas al sistema para permitir a los procesos conectarse a vectores de interrupción y prestar servicio a los sucesos.

Determinados sistemas de tiempo real encastrados, tales como los controladores de herramientas, no disponen de almacenamiento secundario. Sin embargo, cuando aparece, la gestión de archivos de los sistemas de tiempo real debe satisfacer en gran medida las mismas exigencias que en los sistemas de tiempo compartido y otros sistemas de multiprogramación. Entre estas exigencias se incluye la protección y el control de acceso. El objetivo principal de la gestión de archivos en sistemas de tiempo real es generalmente la velocidad de acceso antes que la utilización eficiente del almacenamiento secundario.

2.2.1.3.3 Sistemas operativos combinados.

En la forma que se han presentado aquí, los diferentes tipos de sistemas operativos están dirigidos a satisfacer necesidades de entornos específicos. Sin embargo, en la práctica un entorno determinado puede no encajar exactamente en ninguno de los ambientes descritos. Así, en centros de cómputo universitarios es frecuente encontrar tanto desarrollo interactivo de programas como extensas simulaciones. Por esta razón, algunos sistemas operativos comerciales proporcionan una combinación de los servicios descritos. Por ejemplo, un sistema de tiempo compartido puede soportar usuarios interactivos e incorporar también un monitor de lotes completo. Esto permite que los programas no interactivos con grandes exigencias de proceso sean ejecutados concurrentemente con programas interactivos. La práctica habitual es asignar baja prioridad a los trabajos de lotes y por tanto ejecutar esos procesos únicamente cuando el procesador esté inactivo. En otras palabras, los trabajos por lotes pueden ser utilizados como relleno para mejorar la utilización del procesador, además de proporcionar un

servicio útil en sí mismo. De la misma manera, algunos sucesos críticos, tales como la recepción y transmisión de paquetes de datos por red, pueden ser manejados en tiempo real sobre sistemas que proporcionan servicios de tiempo compartido a los usuarios sobre terminales.

2.2.1.4 Sistemas operativos distribuidos.

Un sistema informático distribuido es una colección de computadores autónomos con capacidad de comunicación y cooperación mediante interconexión hardware y software.

Un sistema operativo distribuido gobierna la operación de un sistema informático distribuido y proporciona una abstracción de máquina virtual a sus usuarios. El objetivo principal de un sistema distribuido es la transparencia. La distribución de los recursos del sistema informático distribuido debe quedar oculta a los usuarios y a los procesos de aplicación; a menos que éstos exijan explícitamente identificarlo.

La compartición global de recursos del sistema tales como la capacidad computacional, los archivos y los dispositivos de E/S es proporcionada por los sistemas operativos distribuidos, facilitando el acceso a recursos remotos, la comunicación con procesos remotos y la distribución computacional. Además de los servicios típicos proporcionados por el SO de un computador (nodo) para los usuarios locales, un SO distribuido proporciona los siguientes servicios adicionales para acceder a los recursos del sistema distribuido: denominación global, sistemas de archivos distribuidos, y facilidades para distribución computacional, tales como comunicación de procesos internodos y llamadas a procedimientos remotos.

2.2.2 PERSPECTIVAS.

Se consideran las siguientes clases de usuarios de los servicios del SO: los usuarios de lenguaje de órdenes y los usuarios de las llamadas al sistema, esquematizados en la figura 2.11. Los usuarios del lenguaje de órdenes son aquellos que obtienen servicios del sistema operativo mediante órdenes que se teclean o pulsan en el terminal o se incorporan en los archivos de comandos para trabajos por lotes. Los usuarios de llamadas al sistema, por otra parte, invocan servicios del sistema operativo mediante llamadas al sistema en tiempo de ejecución. Éstas se encuentran generalmente incluidas en programas y son activadas durante su ejecución.

2.2.2.1 Perspectiva del usuario de lenguaje de órdenes.

Los lenguajes de órdenes suelen ser específicos de cada SO y se facilitan en la interfaz hombre - máquina. El rango y funcionalidad de las órdenes tienen similitud de un SO a otro. Las clases funcionales de las órdenes típicas del sistema operativo se dan en la tabla 2.1.

Tabla 2.1 Ordenes típicas de un sistema operativo por clase funcional.

Clase funcional	**Orden típica**
Conexión y seguridad	INGRESAR, SALIR, CAMBIAR-CLAVE
Control de programas	CARGAR, EJECUTAR, ABORTAR, CORRER
Gestión de archivos	CREAR, ELIMINAR, RENOMBRAR, COPIAR, MOVER, FORMATEAR, VERIFICAR
Informe de estados	LISTAR-PROCESOS, LISTAR-USUARIOS
Administración de sistema	CREAR CUENTA, LISTAR-ERRORES

Las operaciones de conexión en sistemas multiusuario incluyen órdenes para iniciar una sesión y terminar una sesión en el sistema, tales como INGRESAR al sistema y SALIR del sistema, y para manipulación de claves, tales como CAMBIAR CLAVE. Tras abrir una sesión, los usuarios pueden actualizar determinados parámetros de su entorno operativo. Es posible definir las características del terminal de usuario, información que es almacenada por el sistema operativo para referencia adicional y utilización por parte de otros programas. El editor de textos, por ejemplo, puede entrar automáticamente en uno de sus modos, dependiendo del tipo de terminal desde el cual haya sido invocado. Otras órdenes pueden permitir designaciones de dispositivos para el usuario, tal como la unidad de disco en donde los archivos van a ser buscados y guardados.

Las órdenes de control de programas permiten cargar y ejecutar programas. La orden explícita de carga permite transferir los programas a memoria, controlando el lugar en donde el programa va a ser cargado y algunas otras funciones parecidas. Un programa puede ser cargado en memoria principal con la orden CARGAR y ejecutado con la orden EJECUTAR. Estas dos acciones suelen combinarse en la orden CORRER. Debido a la elevada frecuencia de utilización de este servicio, muchos sistemas permiten que los programas sean ejecutados simplemente escribiendo el nombre del

archivo que contiene el programa ejecutable. La ejecución de los programas puede ser forzada a terminar mediante la orden ABORTAR. Algunos SO proporcionan órdenes u opciones para ejecutar programas en un momento específico del día, o repetitivamente a intervalos especificados. Las órdenes para especificar y modificar atributos de programa, tales como la prioridad de planificación, se suelen incluir también en esta clase.

Las órdenes de gestión de archivos proporcionan generalmente facilidades par, creación y mantenimiento de archivos tales como CREAR, ELIMINAR y RENOMBRAR. Los directorios o carpetas pueden ser listados o impresos mediante las órdenes de esta clase. Los archivos también pueden ser copiados, movidos, concatenados e impresos en respuesta a órdenes de usuario. Muchos sistemas operativos disponen de utilidades de comparación y ordenación para procesar el contenido de los archivos. Los volúmenes de almacenamiento suelen ser manipulados por un conjunto de órdenes tales como inicializar, verificar consistencia, detectar errores.

Las órdenes para informar sobre el estado del computador, en principio, permiten obtener información respecto de las actividades iniciadas por el usuario y de los dispositivos asignados o respecto del sistema completo. Así el usuario puede inquirir sobre el estado de sus procesos/hilos, el tamaño y contenido de las colas de impresión o los usuarios activos en el sistema. Determinados sistemas operativos proporcionan informes bastante extensos tales como la actividad de fallos de página o la visualización dinámica de un resumen de la actividad del sistema. Los monitores dinámicos, por su consumo de recursos, pueden estar restringidos a ciertas clases de usuarios.

Las órdenes de administración del sistema proporcionan facilidades para creación y mantenimiento de cuentas de usuario y especificación de restricciones de uso de recursos; normalmente esta restringidas a gestores del sistema y a personal de mantenimiento. Los administradores del sistema suelen obtener informes estadísticos detallados del comportamiento del sistema. Con frecuencia se proporcionan utilidades para análisis de errores acumulados y registrado en periodos de tiempo. Otras órdenes de esta clase permiten la definición y el envío de mensajes y anuncios del sistema.

Además de las clases de órdenes comentadas, existen otras como las que involucran el correo electrónico, el chat, mensajería, protección y seguridad.

2.2.2.2 Perspectiva del usuario de llamadas al sistema.

Los programadores de aplicaciones y sistemas invocan los servicios del SO por medio de llamadas al sistema operativo. Las llamadas soportadas por el SO comúnmente se denomina interfaz del SO o interfaz de programación de aplicaciones (API). Las órdenes emitidas por los usuarios del lenguaje de órdenes suelen ser traducidas y se ejecutan como una serie de llamadas al sistema. Las llamadas al sistema permiten un control más detallado sobre las operaciones del SO y un acceso más directo a las facilidades hardware.

Las llamadas al sistema para ejecución de programas y control de procesos/hilos incluyen generalmente los siguientes servicios: planificación de procesos/hilos en el tiempo, los servicios disponibles mediante las ordenes EJECUTAR y ABORTAR, suspensión de uno o más procesos/hilos hasta que ocurra una condición específica, reanudación de procesos/hilos previamente suspendidos, definición/modificación de atributos de ejecución de programas. En sistemas de tiempo real suelen existir facilidades para comunicación y sincronización entre procesos/hilos. Los procesos/hilos pueden intercambiar datos y señales de sincronización para sincronizar sus ejecuciones con ciertos sucesos

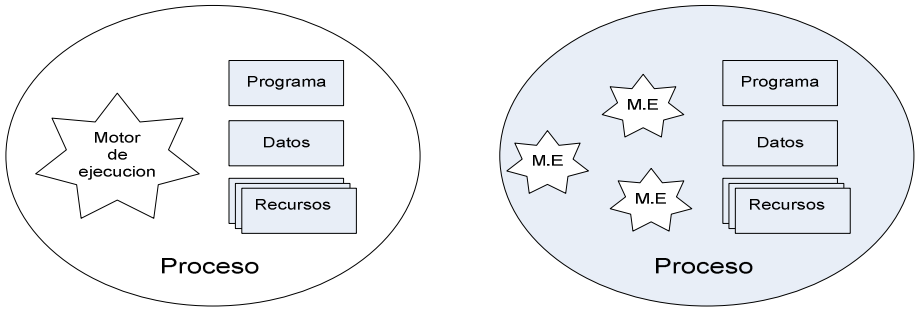
Las llamadas al sistema para la gestión de recursos proporcionan servicios para asignación, reserva y solicitud de los recursos del sistema. Existen llamadas al sistema para extender o reducir la cantidad de memoria asignada al proceso invocante. Otros recursos pueden ser asignados o reservados por el programa invocante y consumidos o devueltos para su custodia por parte del SO.

En sistemas operativos actuales suelen combinarse funcionalmente las llamadas al sistema para gestión de dispositivos y archivos. Básicamente, esta clase de llamadas al sistema proporcionan facilidades para comunicarse con dispositivos de entrada/salida. Además de la lectura y escritura de elementos individuales o de bloques de datos, se proporcionan llamadas tales como ABRIR y CERRAR para gestionar las conexiones lógicas con los dispositivos. Aunque virtualmente todos los dispositivos de E/S pueden ser accedidos en serie, se proporcionan llamadas especiales para acceso aleatorio a dispositivos con estructura de bloques, tales como discos. También se dispone de llamadas al sistema para inicialización de dispositivos y para la selección de modos específicos de operación.

2.2.3 FUNCIONES BASICAS.

El SO como interfaz crea una variedad de componentes abstractos (procesos, hilos, recursos) para la ejecución de programas y como administrador coordina el uso de esos componentes según las políticas de administración del computador.

El sistema operativo gestiona los procesos y los recursos del computador para realizar las tareas de cómputo. Un computo es la combinación de un proceso, al menos un hilo (implícito o explícito) y una colección de recursos. Un proceso se forma a partir del código binario a ejecutar (programa), los datos con los que se ejecutará el programa y los recursos necesarios para la ejecución, representados en la figura 2.13. El proceso es el marco de trabajo en el cual tiene lugar la computación, realizada por el motor de ejecución (elemento activo).



a. Proceso clásico b. Proceso moderno e hilos

Figura 2.13 Esquemas de procesos

Tradicionalmente, los sistemas operativos solo permitían un motor de ejecución por proceso, tal como se esquematiza en la figura 2.13a. En los sistemas operativos actuales, el proceso puede contener múltiples motores de ejecución como se sugiere en la figura 2.13b. El motor de ejecución suele recibir diversas denominaciones entre las que se destacan: proceso ligero, subproceso, hilo, hebra, flujo de control, flujo de instrucciones, flujo de ejecución. Al enfoque tradicional de proceso – hilo implícito se le denomina **proceso clásico** y el planteamiento con múltiples motores de ejecución constituye el enfoque **proceso moderno e hilos**.

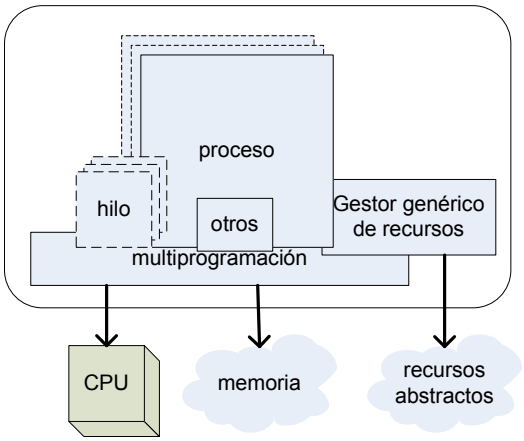
Las abstracciones proceso e hilo son mecanismos fundamentales dentro del SO para gestionar la ejecución concurrente de programas, cada proceso es creado y vive en una máquina abstracta.

Con el devenir del tiempo, las funciones básicas del SO han ido destilándose para satisfacer cada uno de los siguientes requerimientos básicos: gestión de procesos, hilos y recursos, gestión de memoria, gestión de dispositivos y gestión de archivos.

2.2.3.1 Gestión de procesos, hilos y recursos.

Los procesos y los hilos son las unidades básicas de cómputo y los recursos (abstractos) son los elementos del entorno de cómputo que necesita un proceso para que puedan ejecutarse sus hilos.

La gestión de procesos, hilos y recursos (o simplemente gestión de procesos), tal como se muestra en la figura 2.14, se encarga de la creación y administración de los procesos, hilos y recursos. Esta parte del SO también administra el hardware del procesador y gestiona diversos recursos abstractos como los mensajes. Además, colabora con la gestión de la memoria principal y la gestión de los dispositivos de E/S. El siguiente capítulo dedicamos a este tema.



2.14 Gestión de procesos, hilos y recursos

La gestión de procesos e hilos típicamente proporciona los siguientes servicios:

- CREAR para crear un nuevo proceso/hilo
- TERMINAR o SALIR para destruir un proceso/hilo y suprimirlo del sistema.
- ABORTAR para la terminación forzada de un proceso/hilo.
- SUSPENDER o BLOQUEAR para suspender indefinidamente un proceso/hilo.
- RETARDAR o DORMIR para suspender un proceso/hilo durante un periodo de tiempo especificado.
- REANUDAR o DESPERTAR para reanudar un proceso/hilo suspendido.

La gestión de recursos asigna recursos a procesos cuando se solicitan desde un hilo y sigue la pista de dichos recursos hasta que todos los hilos dejan de usarlo; así mismo permite que varios procesos (e hilos) compartan el mismo computador mediante contextos de ejecución múltiple y la planificación del procesador de modo que cada hilo reciba una fracción justa del tiempo disponible.

Las relaciones fundamentales entre los procesos/hilos concurrentes, que comparten los recursos del computador, son de competencia y de colaboración. Todos los procesos/hilos compiten unos con otros por la asignación de los recursos del sistema necesarios para sus operaciones respectivas. Además, los procesos/hilos cooperativos intercambian datos y señales de sincronización.

La gestión de procesos se encarga del aislamiento de los recursos entre los procesos/hilos que compiten y de la compartición de recursos entre los procesos/hilos cooperativos. Para una correcta compartición de recursos entre procesos/hilos, la gestión de procesos brinda mecanismos de sincronización y comunicaciones entre procesos/hilos y maneja los interbloqueos.

2.2.3.2 Gestión de memoria.

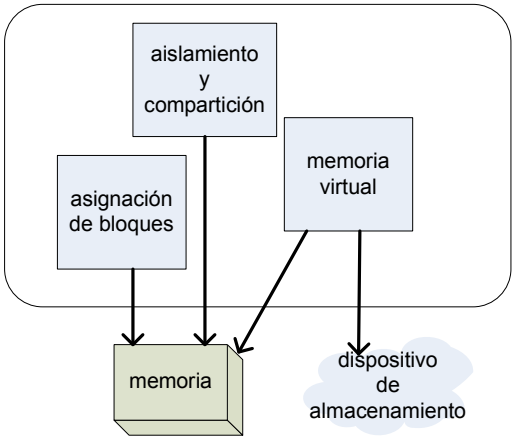
El sistema de memoria incluye las partes del sistema informático que almacenan información, particularmente la UM y los dispositivos periféricos de almacenamiento. La UM o memoria principal, también llamada memoria primaria o de ejecución, aloja la información mientras esta siendo usada por el procesador. Los dispositivos periféricos soportan la memoria secundaria. La memoria principal se referencia byte a byte, tiene un tiempo de acceso corto y suele ser volátil. La memoria secundaria se referencia como bloques de bytes, tiene un tiempo de acceso largo y es persistente. El reto de la programación es mantener los programas y los datos en memoria primaria mientras están en uso por la CPU y almacenar la información en memoria secundaria tan pronto ha sido creada o usada. El logro de este objetivo permitirá que los procesos/hilos hagan un uso eficiente de la memoria y que las pérdidas de información sean mínimas.

Para alcanzar el objetivo planteado, la gestión de memoria se ocupa de la asignación de memoria física de capacidad finita a los procesos que lo solicitan, del aislamiento de la protección de la memoria asignada a cada proceso concurrente y de la compartición de memoria entre procesos cooperativos. La gestión de memoria también

se encarga del mecanismo de memoria virtual de modo que la memoria primaria abstracta puede ser mayor que la memoria primaria física. Estas funciones de la gestión de memoria se esquematizan en la figura 2.15.

Según el tipo de SO, se emplean diferentes estrategias de asignación de memoria a los procesos agrupadas en asignación contigua y asignación no contigua. La asignación contigua aloja el proceso en un área de memoria primaria. En la asignación no contigua, el proceso de descompone en partes, cada una de las cuales se aloja en un área de memoria.

Las estrategias de asignación contiguas pueden ser de particiones fijas y particiones variables. Las últimas utilizan varios algoritmos para encontrar un área libre de memoria apropiada para alojar un proceso, entre los que se destacan: primer ajuste (siguiente ajuste), mejor ajuste, peor ajuste. Estas estrategias pueden mejorarse con la estrategia de intercambio (swapping).



2.15 Gestión de memoria

Entre las estrategias de asignación no contiguas se destacan la segmentación, la paginación y la combinación de ambas. Los segmentos de un proceso, elementos lógicos con distinto contenido (código, datos, pila) y de longitudes diferentes, se alojan en áreas de memoria no contiguas (segmentos de memoria). Las páginas de un proceso, elementos de la misma longitud, se alojan en las páginas de memoria (áreas de memoria primaria no contiguas de la misma longitud).

En un entorno de procesos concurrentes la gestión de memoria debe soportar simultáneamente la protección de memoria, aislando espacios de direcciones (disjuntos) de procesos diferentes, y la compartición de memoria, para permitir que procesos/hilos cooperativos accedan a áreas comunes de memoria.

La memoria virtual integra la memoria primaria del computador con la memoria de los dispositivos periféricos de almacenamiento, permitiendo la ejecución de procesos parcialmente cargados en memoria primaria. La gestión de memoria, por medio del mecanismo de memoria virtual, transfiere automáticamente la información desde la memoria primaria hacia las memorias secundarias y viceversa.

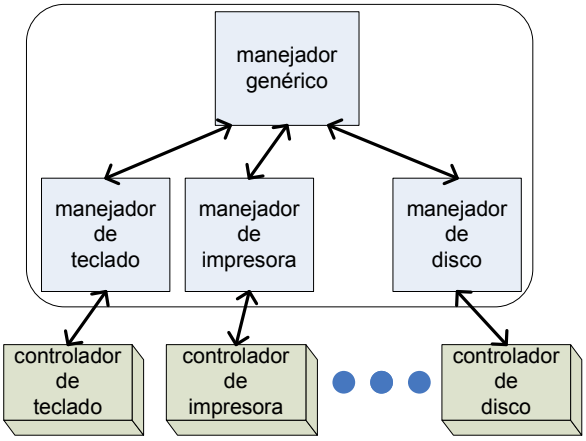
2.2.3.3 Gestión dispositivos.

La mayoría de sistemas operativos abstraen a todos los dispositivos como las cintas, los discos, los terminales y las impresoras de la misma forma; pero proporcionan diferentes aproximaciones especiales para el procesador y la memoria.

El SO distingue entre los dispositivos orientados a bloque y los orientados a caracter. Una operación de E/S sobre un dispositivo orientado a caracter lee o escribe un solo byte. Una operación de E/S sobre un dispositivo orientado a bloque lee o escribe un número determinado de bytes. Muchos de los dispositivos orientados a caracter usan el puerto serial, paralelo, usb o firewire. Algunos dispositivos orientados a bloque permiten acceso secuencial (el bloque i + 1 tras el bloque i), mientras que otros permiten acceso aleatorio a los bloques. Los terminales, las impresoras, los escáneres, los modems, los conmutadores y otros son dispositivos de comunicaciones orientados a carácter. Las cintas magnéticas son dispositivos de almacenamiento orientados a bloque de acceso secuencial. Los discos magnéticos y ópticos son dispositivos de almacenamiento orientados a bloque de acceso aleatorio.

La gestión de dispositivos abstrae los dispositivos, excepto el procesador y la memoria, y gestiona su asignación, aislamiento y compartición. La gestión de los dispositivos concierne a la forma en que se manejan los dispositivos genéricos. Como se aprecia en la figura 2.16, en la gestión de dispositivos hay partes dependientes del dispositivo y partes independientes del dispositivo, manejadores de dispositivo y manejador genérico respectivamente.

Los manejadores de dispositivos implementan los aspectos de gestión específicos de cada tipo de dispositivo. Un manejador de teclado se construye explícitamente para percibir las pulsaciones de teclas provenientes de un teclado según se van pulsando. Un manejador de pantalla se construye de forma que escriba caracteres o gráficos sobre una pantalla específica (CRT o LCD).



2.16 Gestión de dispositivos

El manejador genérico define un entorno software general en el que puede ejecutarse cualquier manejador de dispositivo. El manejador genérico incluye la interfaz de llamadas al SO y un mecanismo para dirigir las llamadas al manejador de dispositivo correcto; presentando llamadas como leer, escribir y otras sobre cualquier dispositivo. El manejador genérico es una parte pequeña de la gestión de dispositivos. La mayoría de la funcionalidad se implementa en los manejadores de dispositivo, que definen las estrategias de E/S.

En este párrafo complementamos algunos aspectos de las estrategias de E/S presentadas en el párrafo 2.1.3. Las estrategias de entrada/salida consideran dos dimensiones: la primera se relaciona con la E/S vía procesador o por DMA y la segunda tiene que ver con el uso de programa (encuesta) o interrupción para indicar la disponibilidad del dispositivo. Estas dos dimensiones definen las siguientes estrategias de entrada/salida: E/S vía CPU con encuesta, E/S por DMA con encuesta, E/S vía CPU con interrupción y E/S por DMA con interrupción. A continuación se presentan aspectos genéricos de la tercera estrategia.

En un escenario de interrupciones, la funcionalidad de la gestión de dispositivos se divide entre los siguientes componentes: manejador iniciador de dispositivo que inicializa el dispositivo, la tabla de estados de dispositivo, el gestor de interrupciones y el manejador de dispositivo. La siguiente relación muestra los pasos para la realización de una instrucción de entrada en un sistema mediante interrupciones:

- El proceso de aplicación solicita una operación de lectura.
- El manejador iniciador consulta el registro de estado para determinar si el dispositivo está libre. Si el dispositivo no estuviera libre, el manejador esperaría.

- El manejador iniciador almacena de entrada sobre el registro de órdenes del controlador, arrancando el dispositivo.
- El manejador iniciador escribe información pertinente en la entrada de la tabla de estados de dispositivo (que contiene una entrada por cada dispositivo del sistema) correspondiente al dispositivo en uso, invoca (opcionalmente) al planificador de procesos y finaliza.
- El dispositivo completa la operación de entrada e interrumpe al procesador, invocando al gestor de interrupciones
- El gestor de interrupciones invoca al manejador de dispositivo que interrumpió.
- El manejador de dispositivo recupera la información pertinente de la tabla de estados de dispositivo.
- El manejador de dispositivo copia el contenido del registro de datos del controlador sobre el espacio del proceso de aplicación.
- El manejador de dispositivo devuelve el control al proceso de aplicación.

2.2.3.4 Gestión de archivos.

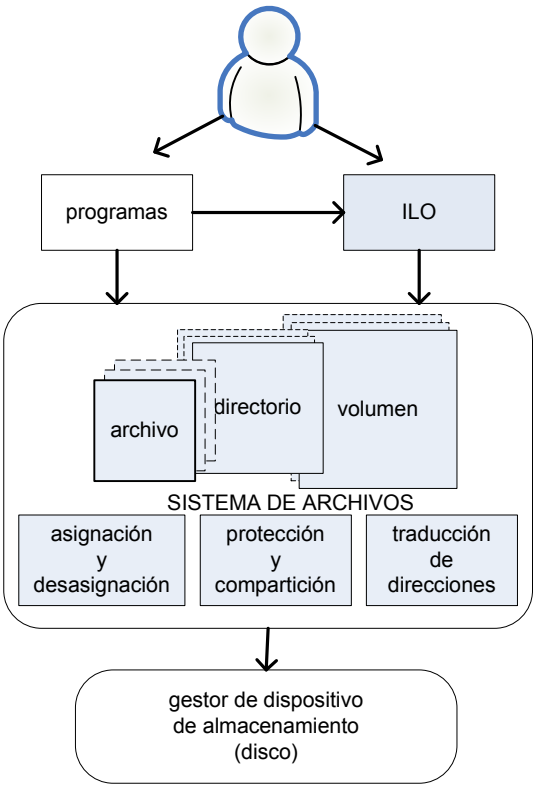
La gestión de archivos es responsable de los archivos. Un archivo está constituido por datos lógicamente relacionados, organizados en una colección identificada por un nombre, y almacenados en memoria secundaria. Un archivo puede contener información de diversa índole: un informe, un programa ejecutable, un grupo de órdenes para el SO, una imagen de un proceso, las páginas de memoria virtual, una gráfico, un audio, un video.

La gestión de archivos amplia la abstracción realizada por la gestión de dispositivos de almacenamiento, presentando un espacio simple y uniforme de archivos con nombre. Un archivo suele presentarse como un arreglo lineal de caracteres o de estructuras de tipo registro.

La gestión de archivos, representada por el sistema de archivos y el intérprete del lenguaje de órdenes (ILO) en la figura 2.17, realiza las abstracciones de archivo, directorio y volumen y proporciona diferentes servicios a programas y a usuarios interactivos. Para ofrecer estos servicios, la gestión de archivos usa los servicios del gestor de dispositivos de almacenamiento y realiza las siguientes funciones: conocimiento de todos los archivos, protección y compartición de archivos, gestión del

espacio en dispositivos de almacenamiento (asignación y desasignación) y traducción de las direcciones lógicas de los archivos a direcciones físicas de los dispositivos de almacenamiento.

El ILO proporciona al usuario interactivo los servicios para manipular archivos, directorios y volúmenes/medios. Ejemplos de manipulación de archivos: crear archivo, copiar archivo, modificar atributos de archivo. Ejemplos de manipulación de directorios: listar directorios, crear directorio, cambiar directorio. Ejemplos de manipulación de volúmenes/medios: formatear volumen/disco, montar unidad, verificar volumen.



2.17 Gestión de archivos

En tiempo de ejecución, la gestión de archivos brinda a los programa servicios tales como: abrir archivo, escribir en archivo, leer de archivo, cerrar archivo. Además los servicios proporcionados por ILO, también pueden ser invocados por los programas.

El sistema de archivos tiene conocimientos de todos los archivos, llevando un inventario de archivos, organizados en directorios (tipos especial de archivo), y alojándolos en volúmenes. El volumen es una visión lógica parcial, total del dispositivo de almacenamiento o de múltiples dispositivos.

La protección de archivos fuerza el aislamiento de archivos, brindando acceso de los usuarios solo a aquellos archivos a los cuales está explícitamente autorizado y del modo especificado, por ejemplo, solo lectura, solo escritura. La compartición de archivos permite a varios usuarios autorizados acceder al mismo archivo concurrentemente.

El sistema de archivos contabiliza el espacio no utilizado del dispositivo de almacenamiento por medio de un depósito de bloques libres. Los bloques necesarios para la creación y crecimiento de archivos se asignan a partir de ese depósito. Cuando se liberan bloques como consecuencia de eliminaciones de archivos o reducciones de sus tamaños, el sistema de archivos los devuelve al depósito de bloques libres (desasignación).

Los programas especifican las partes de los archivos que requieren leer o escribir en términos de direcciones lógicas relativas al archivo. Por otro lado, por ejemplo, el manejador de disco maneja direcciones físicas de disco, especificadas en términos de números de cilindro, de cabeza y de sector. El sistema de archivos debe traducir las direcciones lógicas suministradas por el programa a direcciones físicas de disco necesarias para procesar las peticiones de lectura y escritura. Además, puesto que el sector es la unidad básica de transferencia de disco, el sistema de archivos debe convertir una petición de programa para acceder a un número arbitrario de bytes de un archivo en una petición para acceder a un número entero de sectores de disco.

2.2.4 ARQUITECTURA.

La determinación de los requisitos funcionales del SO, relativos a las abstracciones de recursos y su compartición, a partir de las funciones presentadas en párrafos anteriores, plantea dos cuestiones recurrentes en el desarrollo del software:

- La **eficiencia** del SO al usar los recursos del computador (especialmente el tiempo del procesador y el espacio de memoria principal), maximizando la disponibilidad de los recursos para su uso por las aplicaciones.
- La **exclusividad de uso de los recursos** que el SO debe proporcionar, permitiendo que los recursos de un proceso conserven información sin temor de que la información sea copiada o alterada. Un fallo en aislamiento de recursos es falta crítica del sistema operativo.

Para resolver estas cuestiones los sistemas operativos usan diferentes mecanismos:

- **Modos de procesamiento** para distinguir, con ayuda de hardware (un bit de modo), la ejecución del SO de la ejecución de aplicaciones.
- Implementación de la parte crítica del SO en un **núcleo**, diseñado como un módulo de software confiable que soporta la operación correcta del resto del software.
- Modos que usan los procesos del usuario para **invocar los servicios** del SO. Se usan diferentes formas de pedir los servicios del SO tales como: llamada a procedimiento, llamada a supervisor, interrupciones software, envío de un mensaje a un proceso del SO.

Estos requisitos funcionales del SO se muestran en el diagrama de bloques de la figura 2.18. El diagrama incluye el hardware del computador.

El gestor de procesos crea el proceso y sub procesos y el entorno de ejecución sobre el hardware y usa las abstracciones producidas por los otros gestores al realizar la gestión de recursos, puesto que cada uno de los otros gestores es responsable de alguna clase de recursos.

El gestor de memoria se encarga de la memoria principal. Además, debe interactuar con el gestor de procesos para coordinar la planificación y asignación de memoria y con el gestor de archivos para la entrada/salida de archivos usando búferes.

El gestor de dispositivos maneja los dispositivos hardware e interactúa con el gestor de archivos para almacenar los archivos en los dispositivos de almacenamiento. El manejo de interrupciones exige del acoplamiento entre el gestor de dispositivos y el gestor de procesos.

El gestor de archivos es responsable del almacenamiento secundario manejando las abstracciones de información en los dispositivos de almacenamiento y su entrada/salida a memoria, normalmente mediante búferes.

En el devenir del tiempo se han planteado diferentes arquitecturas de realización (diseño e implementación) de los requisitos funcionales y no funcionales planteados del SO, destacándose las siguientes aproximaciones: monolítica, modular, núcleo

extensible, y estratificada (por capas). A continuación se resume una aproximación de la arquitectura por capas del SO, propuesta por [100] y esquematizada en la tabla 2.2.

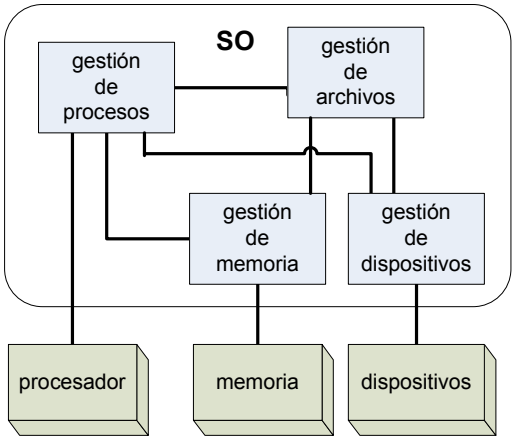


Figura 2.18 Requisitos funcionales de SO.

El nivel 1 es la primera capa del SO que solo utiliza los objetos y las operaciones disponibles en el hardware. La configuración hardware subyacente suele incluir controlador de interrupciones, temporizador de intervalos y soporte para gestión de memoria. Este nivel corresponde al núcleo del SO y gestiona básicamente los procesos. El núcleo lleva la cuenta de los procesos activos mediante estructuras de datos que muestran el estado del sistema. El planificador selecciona qué proceso ejecutar cuando se desactiva el proceso actualmente en ejecución. Al asignar el procesador a un nuevo proceso, el núcleo efectúa una operación de cambio de contexto que incluye salvar el estado del proceso actual y restaurar el estado del siguiente. En este nivel se gestiona las interrupciones enmascarándolas cuándo y cómo sea necesario y proporcionando facilidades para conectar rutinas de servicio a interrupciones hardware. Este nivel también puede proporcionar mecanismos básicos para sincronización y posiblemente comunicación entre procesos, tales como semáforos, monitores o mensajes

El nivel 2 está a cargo de la entrada/salida básica en general, proporcionando facilidades para la gestión de memoria secundaria necesaria para soportar la gestión de memoria principal en el Nivel 3. El nivel 2 permite que se transfieran bloques de datos entre los almacenamientos primario y secundario. Proporciona una abstracción de muy bajo nivel de la memoria secundaria como secuencia lineal de bloques de datos para

fines de lectura y escritura. Las peticiones efectuadas de este modo se traducen por órdenes hardware para mover y posicionar las cabezas del disco.

El nivel 3 gestiona la memoria primaria. Asigna memoria a los procesos al cargarlos y lo libera cuando éstos terminan. El aislamiento de espacios de direcciones y algunas formas controladas de compartición de memoria se soportan en el nivel 3. La memoria virtual, que proporciona la ilusión le disponer de una memoria mayor de que la disponible físicamente, puede implementarse en este nivel. Los módulos del nivel 3 manejan las interrupciones hardware, producidas al direccionar datos que no se encuentran en la memoria principal. En este caso, los bloques de datos ausentes son traídos desde el almacenamiento secundario utilizando facilidades del nivel 2. Si no hay espacio disponible, se desocupa el espacio necesario mediante la retirada temporal de algunos datos llevándolos al almacenamiento secundario.

Tabla 2.2 Capas del Sistema Operativo.

Nivel	Nombre	Objetos	Operaciones típicas
5	Interprete de LO	Datos de entorno	Comandos del lenguaje de órdenes.
4	Gestión de archivos	Archivos, dispositivos	Crear, eliminar, abrir, cerrar, leer, escribir.
3	Gestión de memoria	Segmentos, páginas	Leer, escribir, acceder.
2	E/S básica	Bloques de datos	Leer, escribir, asignar, liberar.
1	Núcleo	Procesos, sub procesos, semáforos	Crear, terminar, suspender, reanudar, señalizar, esperar.

El nivel 4 proporciona facilidades para almacenamiento a largo plazo y manipulación de archivos con nombre. En el nivel 4 se implementan formas más sofisticadas de asignación de espacio y acceso a datos en memoria secundaria que las proporcionadas por el nivel 2. Los archivos, o partes de ellos, pueden ser accedidos y actualizados por medio de órdenes de alto nivel y sin necesidad de especificar los números o direcciones de los bloques de datos tal como requiere el nivel 2. En el nivel 4, la información es generalmente direccionada de una manera relativa a archivos. El nivel 4 también gestiona los dispositivos y periféricos externos, tales como impresoras y terminales. Las diferencias hardware entre diferentes tipos de dispositivos, tales como si están orientados a carácter o estructurados en bloques, son encubiertas por el software a

este nivel para proporcionar una visión uniforme de archivos y dispositivos para los niveles superiores y finalmente para los usuarios del sistema. Este interfaz estándar también puede extenderse con una facilidad de comunicación entre programas denominada cauce, que es esencialmente un canal de comunicación virtual unidireccional. Se pueden escribir flujos de datos en un extremo del cauce y leerlos en el otro utilizando básicamente el mismo grupo de llamadas, tales como ABRIR y LEER, que están disponibles para manipulación de archivos y dispositivos. El soporte para sistemas distribuidos, incluyendo la denominación global y el encaminamiento de peticiones de recursos no locales, se implementa en algún punto intermedio entre los niveles 4 y 5.

El nivel 5 es el intérprete del lenguaje de órdenes. Proporciona el interfaz entre los usuarios interactivos y el SO. Los módulos del nivel 5 utilizan facilidades proporcionadas por los niveles inferiores para aceptar líneas de órdenes desde los terminales. Estas líneas de entrada son entonces analizadas sintácticamente para separar las órdenes de los parámetros e identificar el tipo de servicio solicitado. Las llamadas al sistema en otros niveles se emplean para proporcionar realmente el servicio. Cuando se solicita ejecutar un programa, el software de este nivel crea el entorno de trabajo e invoca a los procesos para realizar el trabajo.

Capitulo III:
GESTION DE PROCESOS.

En un sistema informático, los procesos/subprocesos (P/S) son los elementos computacionales activos que trabajan sobre recursos pasivos como la memoria, los dispositivos, los archivos, los mensajes. La gestión de procesos en el SO proporciona una gama de servicios para definir, soportar y administrar los procesos, subprocesos y recursos.

Este capítulo extiende el tema empezado en el capítulo anterior, párrafo 2.2.3.1, y empieza presentando los elementos principales de la gestión de procesos, para luego tocar aspectos básicos de la planificación, la sincronización y el interbloqueo. La gestión de procesos ha sido extensamente desarrollada por muchos autores, tales como: [5], [12], [14], [16], [17]…; que se toman como base para esta presentación.

3.1 PROCESOS, SUBPROCESOS Y RECURSOS.

A partir del hardware del computador nos aproximamos a la máquina abstracta y aspectos relacionados, en la cual se ejecutan los procesos/subprocesos. Luego contemplamos los servicios a cargo de la gestión de procesos. Después desarrollamos las abstracciones de proceso, subprocesos y recursos.

3.1.1 MAQUINAS ABSTRACTAS.

Los sistemas operativos utilizan el multiproceso para proporcionar a los programas de aplicación la ilusión de que tienen su propia máquina exclusiva (máquina abstracta) para ejecutarse; asignando bloques diferentes de memoria principal a los procesos (multiplexación espacial) y distribuyendo el tiempo del procesador entre ellos (multiplexación temporal). La máquina abstracta es un concepto fundamental en el multiproceso, ya que define el entorno computacional lógico en el que se ejecuta un proceso clásico.

Cada proceso podría ejecutar el código binario en la máquina abstracta que se comportaría exactamente como la máquina real subyacente. La figura 3.1 es una representación pictórica de esta abstracción ideal. Las características de cada máquina abstracta se modelan con el comportamiento de la CPU y la memoria reales en un

computador von Neumann. La unidad de control abstracta del proceso gestiona el programa del proceso de acuerdo con el algoritmo base del proceso hardware descrito en el párrafo 2.1.4. La ALU abstracta ejecuta instrucciones. El programa y los datos para la ejecución se almacenan en la memoria abstracta de ejecución.

El SO se carga en la memoria principal (multiplexado en espacio con los programas de aplicación) y se ejecuta por debajo de la "interfaz del SO". Cuando se arranca el computador, el SO comienza a ejecutarse. Cuando elige ejecutar un proceso en una máquina abstracta, hace que el procesador salte al bloque de memoria principal que contiene el código para la máquina abstracta elegida y comience a ejecutar ese código. Tras un tiempo (marcado por la cesión del procesador o por la ocurrencia de una interrupción), el SO retorna el control y ejecuta su propio código de nuevo. Repitiendo esta secuencia de operaciones, el SO consigue que el hardware simule la actividad de una colección de máquinas abstractas. Esta simulación no siempre será perfecta ya que, por ejemplo, la máquina abstracta no puede ejecutar instrucciones privilegiadas. Los servicios del SO se encargan de las instrucciones privilegiadas.

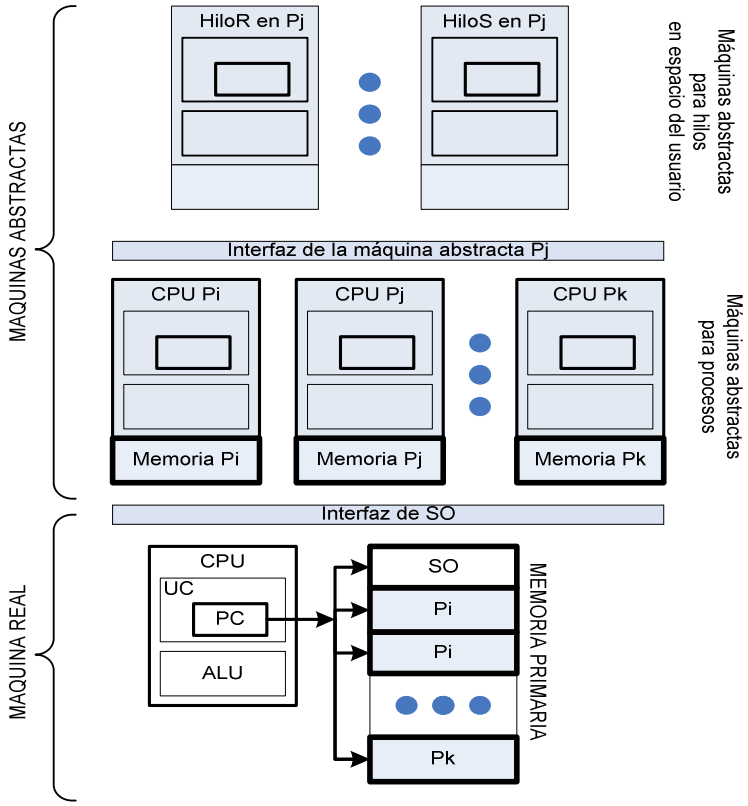


Figura 3.1 Máquinas real y abstractas

Los programas de aplicación invocan los servicios del SO llamando a las funciones en la interfaz de la máquina abstracta (IMA), también referida como la interfaz de llamadas al sistema o interfaz de programación de aplicaciones (API). Las funciones se implementan por las diferentes partes del SO: el gestor de dispositivos, el gestor de procesos, el gestor de memoria y el gestor de archivos.

Un proceso clásico tiene un único subproceso implícito (llamado subproceso base), el cual se ejecuta en una máquina abstracta, que modela el procesador y la memoria abstractos. Cuando el procesador de la máquina abstracta está en ejecución, el subproceso base está en ejecución. Cuando el procesador abstracto se para, el subproceso base se suspende. Un proceso moderno también se aloja en una máquina abstracta; pero permite que más subprocesos compartan sus recursos (procesador abstracto y memoria abstracta).

Conceptualmente, el subproceso puede definirse como una nueva abstracción: ¡Suponga que cada máquina abstracta de proceso, mostrada en la figura 3.1, está diseñada para ser una máquina con multiproceso! La idea es que los subprocesos estén multiplexados en el tiempo sobre el procesador abstracto de proceso (que, a su vez, está multiplexado en el tiempo sobre el procesador real). En esencia, así se implementan los procesos modernos y los subprocesos en el espacio del usuario. El SO subyacente implementa procesos clásicos, y la biblioteca de subprocesos, en el espacio del usuario, se ejecuta sobre la máquina abstracta de proceso para multiplexar los subprocesos dentro del proceso moderno.

Los sistemas operativos modernos que proporcionan un soporte explícito para subprocesos en el núcleo, separan explícitamente la noción de proceso de la de subproceso y gestionan las dos entidades separadas. Por ejemplo, un SO que soporte subprocesos en el núcleo multiplexa en el tiempo la ejecución de los subprocesos en lugar de los procesos.

Los procesos clásicos y los subprocesos modernos son los elementos activos de cómputo, mientras que el proceso moderno y los recursos son los elementos pasivos. Cuando un proceso/subproceso necesita más memoria, un archivo o tiempo del procesador, solicita esos recursos al SO.

Un proceso dispone de una colección de direcciones que puede referenciar, normalmente de bytes, denominada espacio de direcciones de proceso. Estas direcciones

se refieren a posiciones de la memoria principal; aunque también se pueden asociar con otros elementos de la máquina abstracta tales como contenido (bytes) de archivos, registros de dispositivos y otros objetos abstractos. El espacio de direcciones proporciona un mecanismo uniforme por el que un proceso puede referenciar como bytes todos los recursos correlacionados con direcciones.

Cada gestor de recursos es responsable de enlazar direcciones con los elementos direccionables del recurso. Por ejemplo, el gestor del recurso memoria asocia bloques de memoria principal con bloques de memoria en el espacio de direcciones. Si el SO no enlaza la interfaz de un recurso con memoria dentro del espacio de direcciones, entonces el proceso no puede referenciar el recurso. El espacio de direcciones es una parte importante de los mecanismos del SO para la protección de los recursos.

La gestión de procesos crea los procesos, los subprocesos y las abstracciones de recursos. La gestión de procesos es la responsable de proporcionar la IMA para que un programa de aplicación se ejecute como si la máquina abstracta fuera una máquina real. La IMA tiene dos tipos de instrucciones: las instrucciones hardware y las invocaciones a funciones del SO. Cuando un proceso ejecuta una instrucción de máquina abstracta que invoca a una función del SO, el computador pasa a ejecutar código del SO.

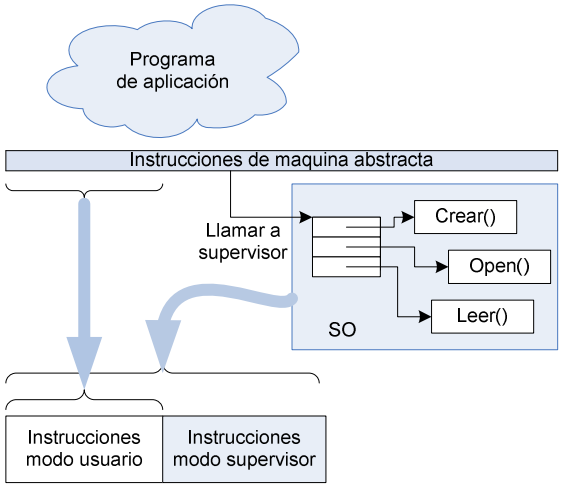


Figura 3.2 Interfaz de la máquina abstracta

En los computadores que disponen de modos (usuario y supervisor), existe la instrucción de lenguaje máquina "llamar a supervisor" (u otras) de modo usuario para gestionar la invocación de funciones del SO. Esta instrucción cambia el procesador a modo supervisor, y a continuación salta al punto de entrada de una función del SO. Esta técnica permite al SO definir una IMA con todas las instrucciones del modo usuario y

todas las llamadas al SO utilizando la instrucción “llamar a supervisor” tal como se muestra en la figura 3.2.

3.1.2 SERVICIOS.

La gestión de procesos implementa los procesos, los subprocesos y la abstracción de recursos en el hardware del computador, controlando la actividad del procesador y de los demás recursos para proporcionar los siguientes servicios de máquina abstracta:

- Creación, administración y terminación de procesos, proporcionando la abstracción de proceso.
- Creación, administración y terminación de subprocesos, proporcionando la abstracción de subproceso.
- Sincronización entre procesos/subprocesos.
- Reserva de recursos, proporcionando la abstracción de recurso (excepto para dispositivos, memoria y archivos).
- Protección de recursos.
- Cooperación con la gestión de dispositivos para implementar la E/S.
- Implementación del espacio de direcciones, cooperando con la gestión de memoria.

Un proceso moderno, entorno de trabajo para los subprocesos, está compuesto de los siguientes elementos:

- Un **espacio de direcciones** a través del cual los programas en ejecución pueden referenciar los recursos direccionables por bytes.
- Un **programa** para definir el comportamiento del proceso.
- Los **datos** utilizados por el proceso. Los subprocesos del proceso comparten los datos.
- Los **recursos** necesarios para la ejecución de los subprocesos. Los subprocesos comparten los recursos asignados al proceso.
- Un **identificador de proceso,** único en todo el sistema.

Cada subproceso tiene las siguientes características:

- Un **ambiente** de trabajo o proceso, en el cual se ejecuta el subproceso.

- Los **datos** específicos del subproceso: PC, pila, variables locales.
- Un **identificador** de subproceso.

Un proceso clásico combina las características de un proceso moderno y un subproceso.

Cada recurso se identifica globalmente en el sistema mediante un **identificador de recurso**. Para cada tipo de recurso se define un tipo de gestor de recurso, y se crean instancias de los mismos para cada instancia de recurso abstracto. Los recursos más fundamentales son el procesador y la memoria.

3.1.3 PROCESO.

La gestión de procesos crea el entorno en que coexisten varios procesos, cada uno en su propia máquina abstracta. Cuando el proceso hardware comience a ejecutar el código del SO, pasará de una máquina abstracta (un contexto) a otra (otro contexto). Estos cambios de contexto pueden suceder cuando el SO obtenga el control del procesador. El SO toma el control del procesador siempre que un proceso/subproceso hace una llamada al sistema o siempre que se produzca una interrupción. Cada proceso/subproceso sólo puede referenciar instrucciones almacenadas en la memoria principal asociada con su espacio de direcciones. La gestión de procesos realiza los cambios de contexto entre procesos/subprocesos.

Cuando se crea un proceso, la gestión de procesos crea una estructura de datos (llamada descriptor de proceso) para guardar todos los detalles necesarios para gestionar el proceso, examina el archivo ejecutable para determinar qué programa debería cargarse en el espacio de direcciones, crea el espacio de direcciones, enlaza las direcciones del programa con el espacio de direcciones y agrega los demás recursos de la máquina abstracta. Entonces, se crea el subproceso base que ejecuta el programa dentro del entorno del proceso. En un SO con procesos clásicos, la gestión de procesos hace esto mediante los campos del descriptor del proceso que representa el subproceso de ejecución. En un SO de procesos modernos, se crea un descriptor de subproceso separado para el subproceso base.

El descriptor de proceso (también denominado bloque de control del proceso, bloque de control de tarea, estructura de tarea) es una estructura de datos en la que el SO

almacena toda la información que necesita para gestionar el proceso, dentro de la que se encuentra la siguiente:

- Nombre interno del proceso, tal como un entero utilizado por el código del SO.
- Estado actual del subproceso base.
- Propietario, identificador interna del propietario como el nombre del usuario.
- Estadísticas de ejecución.
- Subproceso, referencia a la lista de subprocesos asociados a este proceso.
- Lista de procesos relacionados
- Espacio de direcciones, descripción del espacio de direcciones y sus enlaces.
- Recursos, referencia a lista de recursos del proceso.
- Pila, referencia a la pila del subproceso en memoria principal.

Cada descriptor de proceso se reserva cuando se crea el proceso, y se libera cuando el proceso termina. En la mayoría de los sistemas operativos, los descriptores de procesos se reservan de una estructura estática tipo arreglo de descriptores de proceso. La longitud del arreglo establece el número máximo de procesos que el SO puede soportar concurrentemente.

3.1.4 SUBPROCESO.

En entornos con procesos modernos y subprocesos, la gestión de procesos separa la ejecución dinámica de los aspectos estáticos del proceso. Cuando se crea un proceso moderno, también se crea su subproceso base. Cuando todos los subprocesos de un proceso han terminado, entonces se elimina el proceso.

La gestión de subprocesos es la parte de la gestión de procesos que crea y gestiona los subprocesos. Durante su vida, un subproceso pasará por diferentes estados, esperando por un recurso o bien ejecutándose. Las principales tareas en la gestión de un subproceso son:

- Crear, administrar y destruir un subproceso.
- Reservar los recursos específicos del subproceso.
- Gestionar los cambios de contexto del subproceso.

La gestión de procesos realiza estas tareas y contiene los descriptores de subproceso. El descriptor de subproceso es la estructura de datos donde el SO guarda la información que necesita para gestionar el subproceso.

La mayoría de los recursos utilizados por un subproceso se reservan para el proceso asociado en lugar de para el subproceso. Sin embargo, existen algunos recursos que son específicos para cada subproceso tales como el PC, la pila y el almacenamiento privado, que deben estar limitados al espacio de direcciones del proceso. Dentro de la información almacenada en el descriptor de subproceso se encuentra la siguiente:

- Estado, el estado actual del subproceso.
- Estadísticas de ejecución,
- Proceso, referencia al descriptor de su proceso.
- Lista de subprocesos relacionados.
- Pila, referencia a memoria principal de la pila del subproceso.
- Otros recursos, referencia a los recursos específicos del subproceso.

3.1.5 DIAGRAMA DE ESTADOS.

Después de haber creado un proceso/subproceso, el SO tiene a su disposición los descriptores que puede utilizar para realizar un seguimiento del proceso/subproceso. Un diagrama de estados representa los diferentes estados en los que un proceso/subproceso puede estar en distintos momentos, y las transiciones posibles de un estado a otro en el SO. Al administrar los procesos, la gestión de procesos provoca que el proceso/subproceso cambie de estado, por ejemplo, reservando el procesador para el proceso/subproceso, bloqueando el proceso/subproceso hasta que se le asigne un recurso o dejando al proceso/subproceso listo para utilizar el procesador.

La figura 3.3 es un ejemplo de un diagrama de estados básico, en el cual, un proceso/subproceso puede estar en cualquiera de los tres estados: ejecución, listo o bloqueado. En esta diagrama, si un proceso/subproceso en el estado ejecución realiza una solicitud de un recurso que no está disponible, entonces la gestión de procesos lo suspende hasta que se consiga reservar el recurso, modificando el estado del proceso/subproceso a bloqueado. Cuando un proceso/subproceso tiene reservada la memoria y se ha creado, se produce la transición inicio, colocando al

proceso/subproceso en el estado listo, esperando a que la gestión de procesos le asigne el procesador.

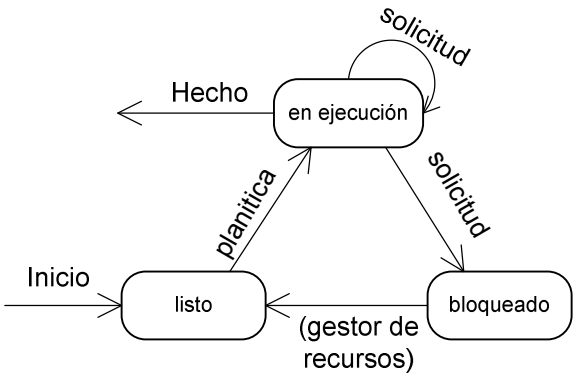


Figura 3.3 Estados de proceso/hilo

La gestión de procesos utiliza el diagrama de estados para determinar el tipo de servicio que proporcionará al proceso/subproceso. Si el proceso está en el estado listo, estará compitiendo por el procesador. La única transición de salida del estado listo es para cambiar el proceso al estado de ejecución cuando se le asigne el procesador. El proceso que se encuentra en el estado ejecución, puede finalizar su ejecución, en tal caso la gestión de procesos liberará los recursos que posea y destruirá el proceso. Si un proceso en ejecución solicita y recibe un recurso sin tener que esperar por él, se le permitirá continuar en el estado ejecución. De lo contrario, la gestión de procesos desasigna el procesador, marca el proceso como bloqueado y notifica a la gestión de recursos que el proceso está esperando por un recurso. Entonces el SO llama al planificador para asignar el procesador a otro proceso seleccionado de entre los del estado listo. Un proceso puede realizar una transición desde el estado bloqueado al de listo cuando la gestión de recursos asigne los recursos solicitados al proceso. Entonces el proceso vuelve a competir por el procesador.

3.1.6 RECURSOS.

Un recurso es cualquier elemento que puede ser solicitado por un proceso, y que puede producir el bloqueo del proceso si no está disponible. Se implementan gestores de recursos para los dispositivos periféricos, el procesador, la sincronización de recursos abstractos, la memoria principal y los archivos. Estos gestores de recursos comparten un comportamiento general, el cual suele aprovecharse para crear nuevos recursos de la máquina abstracta (terminales virtuales, procesadores de aritmética especializada,

mecanismos de procesamiento de cadenas de caracteres, motores gráficos) que se pueden utilizar por los procesos.

El gestor de recurso está compuesto de una parte genérica (común a todos los gestores de recurso) y un comportamiento específico del recurso. La parte genérica del gestor de recursos es un **mecanismo** para reservar recursos y el comportamiento específico del recurso está determinado por una **política**.

El comportamiento general de los gestores de recurso se presenta en la figura 3.4. Un proceso en ejecución solicita recursos abstractos al gestor de recursos por medio de la función solicitar ( ) del SO. Si el gestor de recursos, aplicando una política, decide asignar los recursos al proceso desde su almacén de recursos, entonces actualiza las estructuras de datos para reflejar la asignación, y permite que el proceso prosiga su ejecución. Si el gestor de recursos decide no asignar los recursos, el proceso será bloqueado y colocado en procesos bloqueados hasta que los recursos estén disponibles para él. Un recurso asignado forma parte de la máquina abstracta del proceso. No hay necesidad de encuestas o interrupciones en la máquina abstracta, ya que la E/S se realiza mediante la llamada a una función del SO. El subproceso ejecuta una instrucción de la máquina abstracta ("llamada a supervisor") para iniciar la operación de E/S, y la función no retornará hasta que la operación de E/S haya finalizado.

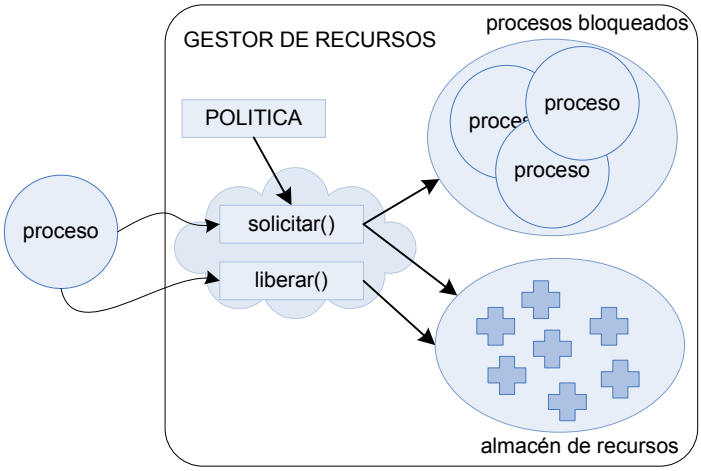


Figura 3.4 Gestor típico de recursos

Cada gestor de recursos mantiene una estructura de datos de descriptor de recursos para los recursos que gestiona. Los detalles del descriptor de recursos dependen del recurso y del SO. La siguiente información suele encontrarse en un descriptor de recursos:

- Nombre interno del recurso, utilizado por el código del SO.
- Unidades totales, de este tipo de recurso en el sistema.
- Unidades disponibles.
- Lista de procesos bloqueados, con solicitudes pendientes de este tipo de recurso.

La noción de recursos se extiende a entidades abstractas tales como mensajes o datos de entrada. Un proceso, que ha solicitado datos de entrada utilizando la operación leer, deberá ser bloqueado hasta que los datos sean "asignados" al proceso tras ser leídos desde el dispositivo.

Los recursos (tales como la memoria) que se pueden reservar y después devolver al sistema una vez que el proceso los ha utilizado se llaman recursos reutilizables. Los recursos más abstractos, como los datos de entrada, que se pueden reservar pero que nunca serán liberados después, son recursos consumibles. El sistema siempre tiene una cantidad fija y finita de unidades de un recurso reutilizable. Sin embargo, el número de unidades de un recurso consumible no está limitado, ya que habrá uno o más procesos productores de cada tipo de recurso consumible. Finalmente, el proceso/subproceso libera los recursos al ejecutar la función del SO liberar ( ) o terminar ( ). La terminación de un proceso produce que el SO libere sus recursos, empezando por la memoria, pero acaba por incluir todos los recursos reutilizables reservados por el proceso. Los recursos consumibles en poder del proceso que termina se asume que han sido consumidos por el proceso, de modo que no se recuperarán.

3.2 PLANIFICACION.

La planificación se encarga de gestionar la compartición del procesador, por multiplexación de tiempo, entre una colección de procesos/subprocesos listos para ejecutarse. El planificador se define a partir de un mecanismo para la conmutación de procesos/subprocesos y una política que determina el orden en el que los procesos/listos reciben servicio.

El estado habitual del computador es el de varios procesos/subprocesos esperando a que el procesador esté disponible. Cuando el procesador está libre, el planificador elige uno de los procesos/subprocesos listos para su ejecución. La política de planificación determina el momento de desalojar un proceso/subproceso del procesador y cuál es el proceso/subproceso listo que lo reemplaza. El mecanismo de

planificación materializa el momento de multiplexar el procesador y realiza el reemplazo de proceso/subproceso en el procesador (conmutación de proceso/subproceso). Desde la perspectiva del subproceso, el planificador hace que el subproceso efectúe una transición desde el estado listo al estado en ejecución, y en algunos casos, del estado en ejecución al estado listo. La figura 3.5 muestra la perspectiva del sistema sobre la planificación de un subproceso en un sistema monoprocesador.

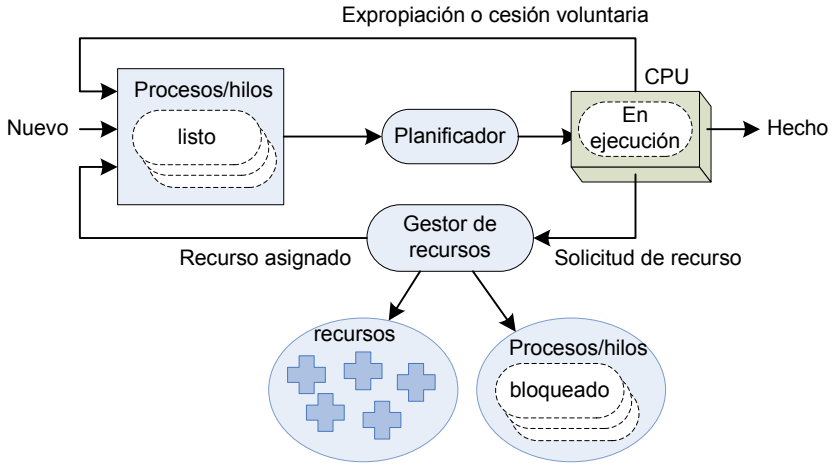


Figura 3.5 Planificación de Procesos/Hilos

Cuando llega un nuevo proceso/subproceso al sistema, se pone en estado listo y se une a la lista de listos del planificador. Para cualquier momento dado, en un computador monoprocesador sólo hay un proceso/subproceso en estado en ejecución. El proceso/subproceso en ejecución puede dejar el procesador por cualquiera de las siguientes razones:

- El P/S completa su función y deja el sistema.
- El P/S solicita un recurso que no está disponible en ese momento. El gestor de recursos bloquea al P/S y lo pone en su lista de bloqueados. Cuando el recurso solicitado esté disponible, el gestor de recursos asignará el recurso al proceso y colocará el P/S en la lista de listos con el estado listo.
- El P/S decide abandonar voluntariamente el procesador y volver al estado listo.
- El P/S abandona involuntariamente el procesador por expropiación. El P/S es retornado a la lista de procesos/subprocesos listos con el estado listo.

En los siguientes párrafos se presenta el mecanismo de planificación y se consideran dos clases de políticas generales: expropiativas y no expropiativas.

3.2.1 MECANISMO DE PLANIFICACION.

El mecanismo de planificación del procesador depende de las características del hardware. La más importante es si el computador tiene o no un dispositivo de temporización. El resto de planificador se realiza por software.

El mecanismo de planificación del procesador comprende múltiples partes, incluyendo el encolador, el selector (despachador) y el conmutador; como se muestra en la figura 3.6.

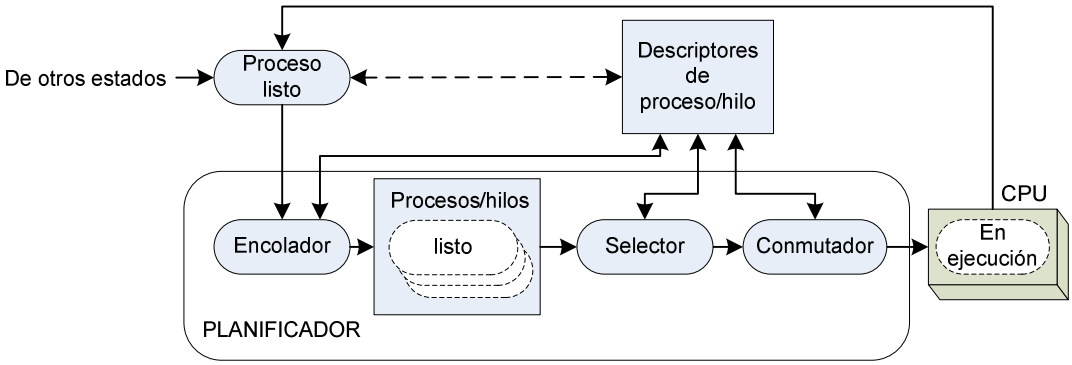


Figura 3.6 Planificador

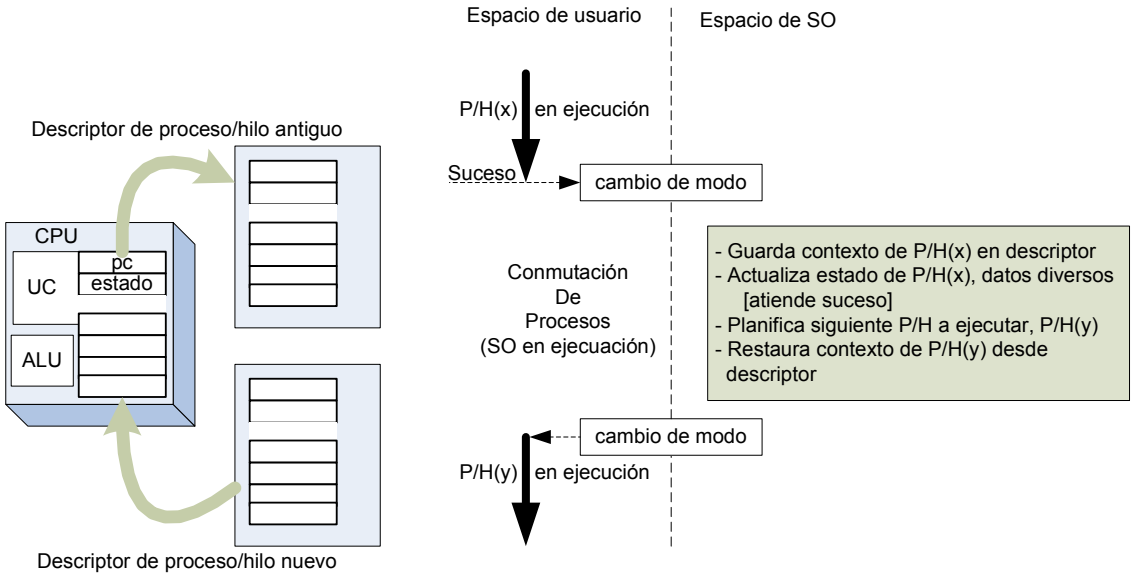
Cuando un P/S transita al estado listo, se actualiza su descriptor y el encolador introduce una referencia a este descriptor en la lista de procesos/subprocesos listos en espera por el procesador.

Cuando el planificador conmuta el procesador de estar ejecutando un P/S a ejecutar otro, el conmutador de contexto guarda, en el descriptor del P/S, los contenidos de los registros pertinentes del procesador para el P/S que se desaloja.

Tras el desalojo del P/S (de aplicación) del procesador, se invoca el selector. (Para que se ejecute el selector, es preciso cargar su contexto en el procesador. El contexto del P/S desalojado es reemplazado (conmutado) por el contexto del selector). El selector selecciona un P/S de la lista de procesos/subprocesos listos y le asigna el procesador mediante una conmutación de contexto desde él mismo hacia el P/S seleccionado.

Cuando el procesador se multiplexa, el P/S "antiguo" se desaloja del procesador y se instala un "nuevo" P/S para que comience a usar el procesador. El procesador contiene diversos registros que alojan los datos y el estado relevante para el P/S actualmente en ejecución. Cuando se detiene la ejecución de un P/S, los contenidos de

los registros de la CPU deben guardarse en el descriptor del P/S, justo antes de que entre el nuevo P/S, para que al reanudarse la ejecución puedan recolocarse dichos contenidos en los mismos registros de la CPU (ver la figura 3.7). La conmutación de contexto afecta significativamente las prestaciones de los computadores que tienen muchos registros generales y de estado que almacenar. El planteamiento de subprocesos en los sistemas operativos modernos es una alternativa para mejorar las prestaciones ante los procesos de los sistemas operativos tradicionales. La inclusión de dos o más conjuntos de registros en el hardware del procesador, es otra alternativa en esta dirección. Uno de los conjuntos sirve cuando el procesador está en modo supervisor y el otro para los subprocesos de modo usuario.



a. Conmutación de contexto

b. Conmutación de Proceso/hilo de usuario

Figura 3.7 Cambios de contextos

Un elemento clave del mecanismo de planificación es la forma en que se invoca al planificador. La aproximación más simple es que el planificador asuma que cada P/S lo invoca explícitamente y de modo periódico, compartiendo así voluntariamente el procesador. Algunos procesadores incluyen la instrucción, ceder, para permitir que un P/S libere la CPU. Este enfoque tiene el problema de depender de la voluntad de cooperar del P/S. Este problema puede evitarse completamente si el propio computador pudiera interrumpir periódicamente el P/S en ejecución (compartición involuntaria).

El sistema de interrupciones puede forzar una interrupción periódica de la ejecución de cualquier P/S. Para eso se incluye un dispositivo temporizador de

intervalo, que genera interrupciones. La figura 3.8 resume el comportamiento de un temporizador de intervalo programable.

```
TemporizarIntervalo ( )
{
  ContadorInterrupcion = ContadorInterrupcion - 1;
  if (ContadorInterrupcion <= 0)
  {
    PeticionInterrupcion = TRUE;
    ContadorInterrupcion = K;
  }
}

PonerIntervalo(int valorProgramable)
{
  K = valorProgramable;
  ContadorInterrupcion = K;
}
```

Figura 3.8 Comportamiento del temporizador programable

El procedimiento TemporizarIntervalo ( ) implementa una interrupción cada K pulsos de reloj y el procedimiento PonerIntervalo ( ) establece el intervalo en K pulsos de reloj. El gestor de interrupción del temporizador invocará al planificador sin que tenga nada que ver el P/S en ejecución. El planificador se asegura de ser llamado al menos una vez cada K pulsos de reloj. Este tipo de planificador se denomina expropiativo.

3.2.2 CRITERIOS DE SELECCIÓN Y RENDIMIENTO.

El tiempo que gasta un P/S en el estado listo hasta que recibe el procesador se determina mediante la política de planificación y el tiempo de conmutación de P/S. En párrafos anteriores se ha visto la importancia de la conmutación de P/S. Las decisiones dirigidas por la política tienen aún más impacto sobre el rendimiento del computador, puesto que pueden incluir intervalos no acotados de tiempo de espera, como en el caso de inanición, donde el P/S se ignora constantemente por el selector y nunca recibe el procesador para completarse.

La política de planificación define los criterios para elegir los procesos/subprocesos que serán ejecutados, basándose en el rendimiento a alcanzarse por el computador. Algunas estrategias favorecen la predecibilidad del rendimiento,

otras favorecen la compartición equitativa, y otras intentan optimizar el rendimiento para cierta clase de procesos/subprocesos.

Entre las medidas de rendimiento y los criterios de selección que los planificadores suelen utilizar en su intento de maximizar el rendimiento del sistema se incluyen:

- Utilización del procesador.
- Productividad.
- Tiempo de retorno.
- Tiempo de espera.
- Tiempo de respuesta.

La utilización del procesador es la fracción de tiempo promedio durante la cual el procesador está ocupado, ejecutando programas de usuario o ejecutando el SO. Una alternativa es considerar únicamente la operación en modo usuario. En cualquier caso, la idea es que, manteniendo el procesador ocupado tanto tiempo como sea posible, la utilización de los demás componentes también será elevada. Sin embargo, al aproximarse al 100% de utilización del procesador, los tiempos de espera promedios y las longitudes de cola promedios tienden a crecer excesivamente.

Un modo de expresar la productividad es por medio del número de procesos/subprocesos de usuario ejecutados por una unidad de tiempo. Mientras mayor sea este número, más trabajo aparentemente está siendo efectuado por el sistema

El tiempo de retorno, R, es el tiempo que transcurre desde la primera aparición del P/S en la cola de procesos/subprocesos listos hasta que es completado por el sistema, y se expresa como el tiempo de ejecución mas el tiempo de espera.

El tiempo de espera, E, es el tiempo que un P/S consume esperando la asignación de recursos. El tiempo de espera puede expresarse como el tiempo de retorno menos el tiempo de ejecución: $E(x) = R(x) - x$; donde x es tiempo de ejecución del P/S, $E(x)$ es tiempo de espera y $R(x)$ es tiempo de retorno.

El tiempo de respuesta en sistemas interactivos se define como el tiempo que transcurre desde el momento que se introduce una orden que desencadena la ejecución de un programa o transacción hasta que empieza a aparecer el resultado en el terminal (tiempo de respuesta de terminal). En sistemas de tiempo real el tiempo de respuesta es

esencialmente latencia, tiempo que transcurre desde la aparición de un suceso hasta que empieza a ejecutarse su P/S asociado (tiempo de respuesta a suceso).

3.2.3 ESTRATEGIAS NO EXPROPIATIVAS.

La planificación no expropiativa permite que cualquier P/S se ejecute hasta su "finalización" una vez que ha recibido el procesador. Esta aproximación es natural en la gestión de procesos/subprocesos porque es intuitiva y, también, se aplica a los sistemas que usan mecanismos voluntarios de invocación del planificador para ceder el procesador. Entre los algoritmos que implementan esta estrategia se encuentran los siguientes: primero en llegar primero en ser servido, siguiente el trabajo más corto, planificación por prioridad, planificación por tiempo limite.

La estrategia **primero en llegar primero en ser servido** (FCFS, first come first served) asigna el procesador a los proceso/subprocesos en el orden de llegada a la cola de procesos/subprocesos listos. El encolador agrega procesos/subprocesos al extremo final de la cola, y el distribuidor remueve proceso/subprocesos de la cabeza de la cola.

La estrategia **siguiente el trabajo más corto** (SJN, shortest job next), también conocida como **primero el trabajo más corto (**SJF, shortest job first), asigna el procesador al P/S que precisa del menor tiempo de servicio. El selector ordena los procesos/subprocesos listos en orden de su tiempo de servicio (ejecución), dándole mayor prioridad a los que requieren menor tiempo de ejecución. Un P/S con largo tiempo de servicio puede sufrir hambre.

En la **planificación por prioridad**, a cada P/S se le asigna una prioridad, y el selector siempre elige el P/S listo con mayor prioridad para asignar al procesador. Las prioridades pueden ser estáticas o dinámicas. La prioridad puede determinarse como función del valor inicial (asignado por el usuario o el sistema), la naturaleza, las necesidades de recursos y el comportamiento en tiempo de ejecución del P/S. En esta estrategia un P/S con prioridad baja puede pasar hambres, problema que suele resolverse con la prioridad de envejecimiento, que crece a medida que el P/S pasa mas tiempo esperando.

La **planificación por tiempos límite** se aplica en los sistemas de tiempo real estricto, donde cada P/S debe completarse antes de la expiración de su plazo (su tiempo límite). El planificador admite un nuevo P/S solo si puede garantizar que será capaz de

proveerle el tiempo de servicio requerido antes del tiempo límite impuesto por los procesos/subprocesos existentes. El **planificador por plazo más inmediato**, una variedad de esta estrategia, asigna al procesador el P/S listo cuyo plazo esté más próximo a cumplirse. El **planificador por mínima laxitud**, otra variedad, selecciona el P/S listo con menor diferencia entre su plazo especificado y su tiempo restante de ejecución.

3.2.4 ESTRATEGIAS EXPROPIATIVAS.

La planificación expropiativa asigna el procesador al P/S de mayor prioridad. Todos los procesos/subprocesos de menor prioridad, incluyendo el que está en ejecución (por interrupción), ceden su sitio al de mayor prioridad cuando éste solicite el procesador.

El planificador es invocado cada vez que un P/S entra en el estado listo como consecuencia de diferentes sucesos, incluyendo la interrupción del dispositivo temporizador al terminar un cuanto de tiempo establecido.

Las estrategias expropiativas se usan para garantizar un tiempo de respuesta rápido para los proceso/subprocesos de alta prioridad o para garantizar una compartición justa del procesador entre todos los procesos/subprocesos. Las estrategias de planificación SJN y de planificación por prioridad; también tienen sus versiones expropiativas, las cuales mantienen siempre en ejecución el P/S de mayor prioridad. La versión expropiativa de la planificación por prioridad también se denomina planificación guiada por sucesos.

Las estrategias expropiativas tienden a tener muchos más cambios de contextos (y más sobrecarga) que las estrategias no expropiativas. Entre las estrategias expropiativas se encuentran las siguientes: turno rotatorio, colas de niveles múltiples, colas de niveles múltiples realimentadas.

En la **planificación por turno rotatorio** o **reparto de tiempo** (RR, Round Robin), el tiempo del procesador se divide en cuotas o quantos que son asignados a los procesos/subprocesos listos. Ningún P/S puede ejecutarse durante más de una cuota de tiempo cuando hay otros esperando en la lista. Si un P/S necesita más tiempo para completarse, después de agotar su quanto, se coloca al final de la cola de procesos/subprocesos listos para esperar una siguiente asignación. Esta reordenación de

la lista de listos tiene el efecto de rebajar la prioridad de planificación del P/S expropiado. El quanto o cuota de tiempo se define por medio del dispositivo temporizador programable. Esta estrategia se aplica en sistemas interactivos de tiempo compartido.

Las **colas de niveles múltiples** (MLQ, Multiple Level Queues) constituyen una extensión de la planificación por prioridad, en la que los procesos/subprocesos listos, según sus características, se sitúan en diferentes colas. Cada cola es atendida entonces por la estrategia de planificación más adecuada. Una posibilidad podría ser organizar tres colas: procesos/subprocesos del sistema e interrupciones de dispositivos (prioridades mayores) con planificación guiada por sucesos, programas interactivos (prioridad intermedia) atendidos por RR y trabajos de lotes (prioridades mínimas) que usen FCFS o SJF. El procesador se distribuye entre las colas usando, por ejemplo, prioridades absolutas o cuotas de tiempo que reflejen la prioridad de los procesos dentro de las colas. En las prioridades absolutas, la cola de menor prioridad se atiende cuando la cola de mayor prioridad está vacía y la cola de mayor prioridad desaloja a la cola de menor prioridad. La planificación frontal/posterior (foreground/background) es un caso de MLQ.

En las **colas de niveles múltiples realimentadas**, los procesos/subprocesos pasan de una cola a otra dependiendo de su comportamiento en tiempo de ejecución. La realimentación en el mecanismo MLQ tiende a clasificar los procesos/subprocesos dinámicamente de acuerdo con la cantidad observada de servicio obtenido, dando preferencia a aquellos que han recibido menos.

3.3 SINCRONIZACION.

Los procesos/subprocesos concurrentes cooperativos esencialmente interactúan de las siguientes formas:

- Compartiendo recursos reutilizables en serie tales como las variables compartidas para lectura/escritura, las impresoras y otros dispositivos. Un recurso reutilizable en serie sólo puede ser utilizado por un P/S cada vez. Estos recursos pueden resultar corrompidos si son manipulados concurrentemente y sin sincronización por más de un P/S.

- Intercambiando señales de temporización para coordinar su progreso colectivo. La señalización es una forma rudimentaria pero habitual de sincronización.
- Comunicándose para intercambiar datos, transmitir información sobre sus progresos respectivos y acumular resultados colectivos. Un medio sencillo de comunicación es una memoria compartida, con acceso sincronizado para los procesos/subprocesos participantes.

En los siguientes párrafos, después de introducir la naturaleza de la sincronización, se presenta el semáforo (un mecanismo de sincronización) con su soporte hardware y algunas de sus abstracciones, incluyendo monitores.

3.3.1 SECCION CRÍTICA Y EXCLUSION MUTUA.

En el tráfico vehicular terrestre, los cruces son partes especiales de una vía, ya que cada uno es compartido entre dos vías diferentes (ver figura 3.9). En este dibujo, un tractor va por una calle mientras un auto se desplaza por la otra. Si el tractor y el auto coinciden en el cruce, se producirá una colisión (fallo del tráfico). El cruce es una sección crítica de cada calle: resulta perfectamente aceptable para el tractor o el auto utilizar el cruce si nada más está pasando por él.

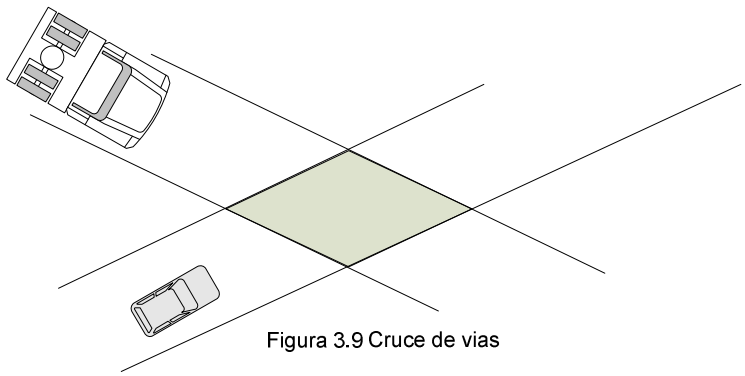


Figura 3.9 Cruce de vias

En los procesos/subprocesos concurrentes se producen secciones críticas, cuando acceden a recursos comunes que comparten. Como los cruces para el tractor y el auto, existen ciertas partes de los procesos/subprocesos que no deberían ejecutarse concurrentemente. Tales partes del código son secciones críticas software. Por ejemplo, si el P/S P1, que gestiona los depósitos de una cuenta de ahorros, y el P/S P2, que gestiona los retiros de dicha cuenta, se ejecutan concurrentemente; ambos necesitan acceder a la variable saldo de la cuenta en diferentes momentos (acceder a saldo es análogo a entrar en un cruce). Este código hace lo que se espera la mayoría del tiempo:

P1 suma a saldo en una operación de depósito y P2 resta de saldo en una operación de retiro. Sin embargo, se producirá un desastre si los dos procesos/subprocesos acceden a la variable saldo concurrentemente. El siguiente esquema muestra el modo en que los subprocesos referencian a saldo compartido:

- en un hipotético lenguaje de programación general:

  double saldo; *// variable compartida*

código para P1	**código para P2**
. . .	. . .
saldo = saldo + deposito;	*saldo = saldo – retiro;*
. . .	. . .

- en un hipotético lenguaje máquina:

código para P1	**código para P2**
carga R1, saldo	*carga R1, saldo*
carga R2, deposito	*carga R2, retiro*
suma R1, R2	*resta R1, R2*
almacena R1, saldo	*almacena R1, saldo*

El desastre ocurre cuando tiene lugar el escenario presentado en la figura 3.10, se perderá la actualización de saldo realizada por P2.

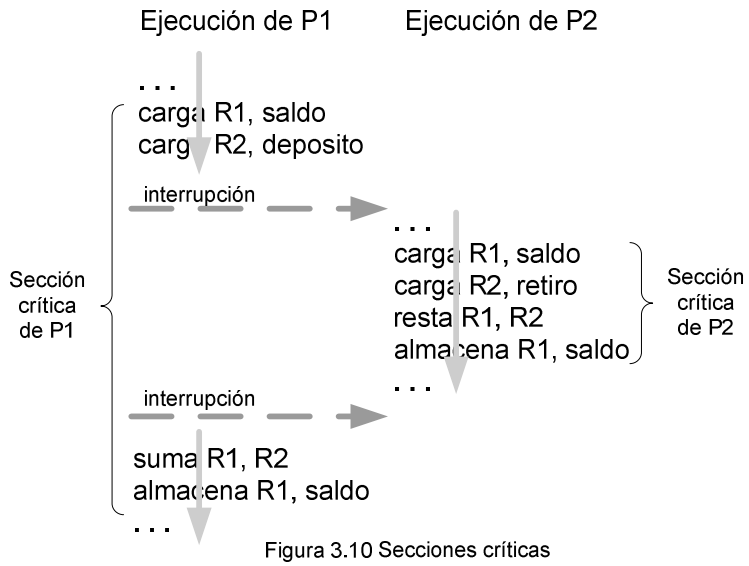


Figura 3.10 Secciones críticas

Los programas que definen P1 y P 2 tienen cada uno una **sección crítica** relacionada con la variable compartida saldo. Para P1, la sección crítica calcula la suma de saldo y depósito, y para P2, la sección crítica calcula la diferencia de saldo y retiro. La ejecución concurrente de los dos subprocesos es **no determinista**, ya que con los mismos datos de entrada puede producir resultados diferentes. Existe una condición de competencia entre P1 y P2, ya que el resultado depende de los tiempos relativos de ejecución de sus respectivas secciones críticas. Cuando los dos subprocesos ejecuten concurrentemente sus secciones críticas respectivas, el resultado del cómputo puede ser erróneo.

No se puede detectar el problema de una sección crítica (o condición de competencia) considerando solo el programa de P1 o de P 2. El problema está en la compartición temporal de la variable saldo, no en el código de cada programa, y se resuelve con exclusión mutua.

La **exclusión mutua**, consiste en permitir que un P/S entre en su sección crítica solamente cuando ningún otro se encuentre concurrentemente en su respectiva sección crítica relacionada; es decir sincroniza el acceso de los procesos/subprocesos a los recursos compartidos, como variables, archivos, búferes y dispositivos. En el tiempo, sólo un P/S a la vez puede utilizar un recurso.

Una solución aceptable al problema de la sección crítica debería:

- Permitir solo a un P/S a la vez ejecutar su sección crítica (exclusión mutua).
- No hacer suposiciones respecto a las velocidades y prioridades relativas de los procesos/subprocesos en competencia.
- Garantizar que la terminación/aborto de cualquier P/S fuera de su sección crítica no afecte la capacidad de los restantes procesos/subprocesos contendientes para acceder al recurso compartido.
- Permitir que sea el conjunto de procesos/subprocesos que desean entrar en sus secciones críticas, él que decida cuál será el P/S que entre en su sección crítica.
- Cuando más de un P/S desee entrar en su sección crítica, conceder la entrada a uno de ellos en tiempo finito.
- Una vez que un P/S pide entrar en su sección crítica, conceder el ingreso en tiempo finito.

En las estrategias de exclusión mutua, cada P/S suele observar el siguiente protocolo:

Negociación del protocolo	*(el ganador continúa)*
Sección crítica	*(uso exclusivo del recurso)*
Protocolo de liberación	*(abandono de la exclusividad)*

Los mecanismos de sincronización de acceso a recursos compartidos deben generalizarse en mecanismos, donde un P/S debería ser capaz de bloquearse hasta que ocurra, en otro P/S, algún **evento** previamente definido (**señalización**). El semáforo viabiliza la exclusión mutua y la señalización Los siguientes fragmentos de código presentan un escenario que requiere señalización.

```
entero x, y;     //variables compartidas(señales)
pA ( )                                  pB ( )
{                                       {
    while (TRUE)                            while (TRUE)
    {                                       {
        <computo A1>;                           leer (x);       //consumir
        escribir (x);    //producir             <computo B1>;
        <computo A2>;                           escribir (y);   //producir
        leer (y);        //consumir             <computo B2>;
    }                                       }
}                                       }
```

El objetivo de estos fragmentos de código es que pB no debería ejecutar su primera instrucción leer ( ) hasta que pA no acabe de escribir ( ) sobre la variable x. Esta sincronización debería producirse una vez por ciclo. Cuando comienza pB, debería suspender su funcionamiento hasta que se produzca el evento escribir (x) en pA. Ésta es una variante del problema de la sección crítica en el que se necesita la sincronización para la cooperación entre pA y pB en ejecución, en lugar de regular el acceso a una sección crítica.

3.3.2 SEMAFOROS.

En los cruces de calles concurridos se soluciona el problema de tráfico utilizando semáforos para coordinar la intersección compartida. En gestión de procesos,

un semáforo es un tipo de dato abstracto del SO que realiza una operación similar a la de los semáforos del tráfico. El semáforo sólo permitirá controlar el recurso compartido a un P/S mientras los demás esperan a que se libere.

Un semáforo de Dijkstra, s, es una variable entera no negativa manipulada sólo por las operaciones primitivas P y V, excepto posiblemente en su inicialización. Estas primitivas para el semáforo con espera activa se definen del modo siguiente:

- **P(s)**; en tanto semáforo sea mayor que 0, comprueba y decrementa semáforo en una operación indivisible o atómica. Sin embargo, si semáforo es igual a 0, el proceso ejecutando la operación P puede ser interrumpido cuando ejecuta la instrucción esperar en el bucle.
- **V(s)**; incrementa semáforo en una operación indivisible o atómica.

La lógica de las versiones de las operaciones P y V con espera activa aparece en la figura 3.11.

```
while(s = = 0) {esperar};
s = s – 1;
```

a. P(s)

```
s = s + 1;
```

b. V(s)

Figura 3.11 Implementación de P y V con espera activa

La operación P se emplea para comprobar de manera indivisible una variable entera y para bloquear el proceso invocador si la variable no es positiva. La operación V lanza indivisiblemente una señal a un proceso bloqueado para permitir1e reanudar su funcionamiento. En la figura 3.12 se presentan ejemplos de sincronización usando semáforos.

El ejemplo de la figura 3.12.a, resuelve el problema de compartición de saldo de la cuenta de ahorros, presentado en el párrafo anterior, usando exclusión mutua. El valor inicial del semáforo mutex es 1. Cuando un P/S está listo para entrar en su sección crítica aplica la operación P sobre mutex. El primer proceso en invocar P sobre mutex pasa y el segundo se bloquea. Cuando el primero invoca V sobre mutex, continúa su ejecución, haciendo que el segundo pueda proceder cuando tome control del procesador.

El ejemplo de la figura 3.12.b, presenta una colaboración entre procesos/subprocesos por medio de señalización planteado en el párrafo anterior. phA y phB necesitan coordinar su actividad de modo que phB no intente leer x hasta que phA

haya escrito en ella. Del mismo modo, phA no debería intentar leer y hasta que phB haya escrito un valor nuevo en y. El P/S phA utiliza el semáforo s1 para lanzar una señal a phB, y el P/S phB utiliza s2 para señalar al phA.

```
  semaforo mutex;
  double saldo; //variable compartida

phDepositar ()
{
  . . .
// entrar sección critica
  P(mutex);
    saldo = saldo + deposito;
// salir sección critica
  V(mutex);
  . . .
}

phRetitar ()
{
  . . .
// entrar sección critica
  P(mutex);
    saldo = saldo - retiro;
// salir sección critica
  V(mutex);
  . . .
}

phPrincipal ()
{
  mutex = 1; //libre
  iniciar phDepositar, phRetirar;
}
```

a. Exclusión mutua

```
  semaforo s1, s2;
  entero x, y; //variables compartidas

phA ()
{
  while(TRUE)
  {
    <computo A1>;
    escribir(x); //producir
    V(s1); //señalar phB
    <computo A2>;
// esperar señal de phB
    P(s2);
    leer(y); //consumir
  }
}

phB ()
{
  while(TRUE)
  {
// esperar señal de phA
    P(s1);
    leer(x); //consumir
    <computo B1>;
    escribir(y); //producir
    V(s2); //señalar phA
    <computo B2>;
  }
}

phPrincipal ()
{
  s1 = 0; s2 = 0;
  iniciar phA, phB;
}
```

b. Señalización

Figura 3.12 Sincronización con semáforos

Un semáforo que solo toma los valores 0 (OCUPADO) y 1 (LIBRE) se denomina **semáforo binario**. Para los semáforos binarios, la lógica de P debería

interpretarse como la espera hasta que la variable semáforo sea igual a LIBRE, seguido de su modificación indivisible para que indique OCUPADO antes de devolver el control al invocador. La operación P implementa, por tanto, la fase de negociación del protocolo de exclusión mutua. V pone el valor a la variable semáforo a LIBRE y representa por tanto la fase de liberación en el protocolo de exclusión mutua.

Un **semáforo general** puede tomar cualquier valor entero. La lógica de las operaciones P y V presentadas en la figura 3.11 es aplicable tanto a semáforos binarios como a semáforos generales.

3.3.3 SOPORTE HARDWARE DE SEMAFOROS.

La definición de semáforos propuesta en la sección anterior se apoya en ciertas y peculiares exigencias de indivisibilidad. La materialización de esta indivisibilidad, y de los semáforos y la exclusión mutua, se realiza finalmente a nivel hardware del computador y se manifiesta en la disponibilidad de determinadas instrucciones del lenguaje máquina de ejecución indivisible tales como: habilitar y deshabilitar interrupciones, comprobar y fijar, comparar e intercambiar.

Habilitar interrupciones (EI) **y deshabilitar interrupciones** (DI) están disponibles en prácticamente todos los computadores. El uso de estas instrucciones para implementar la exclusión mutua en sistemas de multiprogramación que operan sobre un monoprocesador sigue la siguiente secuencia:

```
DI                  ; deshabilitar interrupciones
sección crítica     ; usar el recurso protegido
EI                  ; habilitar interrupciones
```

La instrucción comprobar y fijar (TS, Test and Set) está pensada como soporte hardware directo de la exclusión mutua. La idea básica es tener una variable de control global con valor LIBRE cuando esté disponible el recurso compartido protegido y con valor OCUPADO en caso contrario. En principio, TS tiene un operando, la dirección de la variable control o registro que puede actuar como semáforo,

```
TS operando         ; comprobar y fijar operando
```

y funciona del modo siguiente:

1. Comparar el valor de operando con OCUPADO y establecer la condición que refleje el resultado.

2. Fijar el operando a OCUPADO.

Estos pasos se ejecutan en una única operación indivisible. La operación P sobre la variable semáforo S, trasmitida por el P/S invocador como argumento de P; puede implementarse mediante el siguiente código:

```
P:    TS     S      ; solicitar acceso exclusivo
      BNF    P      ; (Branch if Not Free) repetir hasta que sea concedido
      RETURN        ; proseguir a la sección crítica
```

Otras instrucciones con ciclo de lectura-modificación-escritura indivisible en sistemas monoprocesadores como INCREMENTAR MEMORIA e INTERCAMBIAR MEMORIA Y REGISTRO también son candidatas para implementar semáforos, puesto que permiten examinar el contenido de una variable y fijarla posteriormente en una operación indivisible tal como lo hace TS.

Para que TS funcione en entornos multiprocesadores debe implementarse un ciclo indivisible de lectura-modificación-escritura sobre el bus del sistema que gobierne la memoria compartida. Algunas familias de computadores brindan instrucciones para entornos multiprocesadores con ciclo indivisible de lectura-modificación-escritura diseñadas para implementar semáforos.

La instrucción comparar e intercambiar (CS, Compare and Swap) no está pensada para implementar directamente operaciones de semáforo, sino para la actualización consistente de variables globales en presencia de actividad concurrente, y sin ningún soporte por parte del sistema operativo o del mecanismo de interrupciones. La instrucción CS es muy conveniente y eficaz para el caso de actualizaciones simples de variables compartidas.

CS tiene tres operandos: un registro que contiene el valor de la variable global en la cual se basa la actualización tentativa (viejoReg), un registro que contiene la actualización tentativa (nuevoReg) y la dirección de la variable global en cuestión (varGlob).

```
CS viejoReg, nuevoReg, varGlob          ;comparar e intercambiar
```

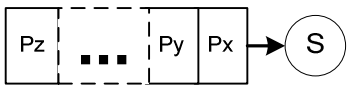
CS consta de la siguiente secuencia de pasos, que se ejecutan como una operación única indivisible.

1. COMPARAR viejoReg con varGlob y FIJAR condición

2. SI (viejoReg == varGlob) ENTONCES varGlob <= nuevoReg
   SI NO viejoReg <= varGlob

3.3.4 SEMAFOROS CON COLAS.

La implementación de los semáforos con espera activa tiene dos importantes desventajas: el potencial aplazamiento indefinido y el consumo de ciclos de procesador por los procesos/subprocesos bloqueados. Tanto el bloqueo activo como la ineficaz espera activa pueden ser aliviados por la implementación de semáforos con colas. La figura 3.13 muestra la estructura de datos y las operaciones del semáforo en la implementación con colas. A cada variable semáforo se le asocia una cola de descriptores de procesos/subprocesos.



a. Estructura del semáforo

si(s = = 0) suspender invocador en s;
en caso contrario s = s – 1;

b. P(s)

si(cola de s no vacía) reanudar un P;
en caso contrario s = s + 1;

c. V(s)

Figura 3.13 Implementación de semáforos con colas

Cuando el recurso solicitado no está disponible, la operación P(s) suspende al P/S solicitante e inserta su descriptor en la cola asociada al semáforo, en caso contrario el semáforo se decrementa y el P/S ingresa a su sección crítica. Un P/S suspendido no consume ciclos del procesador. Un P/S suspendido ya no puede proseguir para probar el semáforo e ingresar a su sección crítica cuando el recurso solicitado quede finalmente disponible. La operación V(s) despierta un P/S que espera en la cola del semáforo, y solo cuando no hay ninguno incrementa la variable semáforo. El problema del bloqueo activo de los semáforos con espera activa puede ser eliminado mediante la elección adecuada de la disciplina de servicio de la cola, por ejemplo con FIFO.

3.3.5 ABSTRACCIONES DE SEMAFOROS.

Los semáforos capturan la esencia de la sincronización entre procesos/subprocesos, aunque las soluciones basadas en semáforos de problemas

complejos de sincronización pueden ser bastante complejas. Los semáforos han sido criticados, principalmente, alrededor de dos temas principales:

- Los semáforos no son estructurados, la sincronización y la integridad del sistema dependen de la adherencia a los protocolos de sincronización específicos.
- Los semáforos no soportan abstracción de datos, no pueden restringir el tipo de operaciones sobre recursos compartidos efectuadas por procesos/subprocesos que han recibido derechos de acceso.

En este escenario se han inventado diversas abstracciones de semáforo. Estas abstracciones no son más potentes que los semáforos, aunque son más sencillas de usar. Se consideran algunas de las abstracciones más importantes: sincronización simultánea, eventos y monitores. Más adelante se presenta la comunicación entre procesos/subprocesos, que puede verse como otro nivel de generalización de sincronización. Todos los sistemas operativos actuales proporcionan semáforos (o una o más abstracciones de semáforos) y un mecanismo de intercomunicación entre procesos/subprocesos.

3.3.5.1 Sincronización simultánea.

En muchos programas concurrentes, los procesos/subprocesos necesitan sincronizarse bajo un conjunto de condiciones en lugar de bajo una única condición. Por ejemplo, dos recursos compartidos, R1 y R2 pueden ser accedidos por una comunidad de procesos/subprocesos, $\{p_j\}$. Algunos de estos procesos sólo necesitan R1, otros sólo necesitan R2 y terceros pueden necesitar un acceso exclusivo a R1 y a R2 a la vez.

```
Ps(semáforo S, entero N)
{
  L1:
    if((S[0]>=1)&& … (S[N-1]>=1))
    {
      for(i=0; i<N; i++) S[i]--;
    }
    else
    {
      Encolar el P/H invocador
      en el primer S[i] < 1;
      goto L1;
    }
}
```

a. Ps(S1, …, Sn)

```
Vs(semaforo S, entero N)
{
  for(i=0; i<N; i++)
  {
    if(procesos/hilos en cola S[i])
    {
      desencolar P/H de cola S[i];
      (este P/H está listo para ejecu-
        tarse, pero podría bloquearse
        de nuevo en Ps)
    }
    else
    {
      S[i]++;
    }
  }
}
```

b. Vs(S1, …, Sn)

Figura 3.14 Sincronización simultánea

En la sincronización simultánea, Ps(S1, ..., Sn) es una operación abstracta simple para obtener todos los semáforos al mismo tiempo, o ninguno de ellos en absoluto. La operación bloquea al P/S invocador cuando no pueda conseguirse alguno de los semáforos. Vs(S1, …, Sn) es una operación abstracta simple que elimina procesos/subprocesos de las colas de semáforos cuando se invoca. Las operaciones Ps y Vs se definen en la figura 3.14.

3.3.5.2 Eventos.

Los eventos se usan para comunicar a un conjunto de procesos/subprocesos la ocurrencia de alguna condición (señalización de semáforos). Si un P/S necesita sincronizar su operación a partir de la ocurrencia de un evento, podrá bloquearse a sí mismo hasta que algún otro elemento del sistema publique el evento. Un evento es análogo a un semáforo, esperar por un evento es análogo a la operación P, y publicar la ocurrencia de un evento es análogo a la operación V.

Un evento se representa por un descriptor de evento u otro nombre parecido (estructura de datos). Un P/S espera por un evento, registrándose en la cola de procesos/subprocesos correspondiente al descriptor de evento. Cuando un P/S publica el evento, la llamada al sistema usa un descriptor de evento para activar uno o más de los procesos/subprocesos bloqueados.

Los nombres de evento (o referencia) suelen definirse en un espacio global de nombres de forma que puedan usarse en todos los subprocesos. Hay típicamente tres funciones miembro en un evento:

- La operación wait( ) bloquea el P/S invocador hasta que otro realice una operación signal ( ).
- La operación signal( ) reanuda un subproceso en espera de los que están suspendidos en el evento por una llamada wait( ). Si no hay subprocesos esperando, signal( ), se ignora.
- La operación queue( ) devuelve el número de procesos/subprocesos que están esperando un evento.

Una gran diferencia entre los eventos y los semáforos es que si no hay algún P/S esperando cuando se eleva una señal, el resultado de la operación signal( ) no se guarda y su ocurrencia no tendrá efecto alguno. La razón de estas semánticas es que una señal

podría representar la situación de que acabara de ocurrir un evento, no que haya ocurrido en algún momento del pasado. Si otro subproceso detecta esta ocurrencia en un tiempo arbitrario posterior (como ocurre con las operaciones pasivas de semáforo), las relaciones causales entre las llamadas a las funciones signal ( ) y wait( ) se perderían.

3.3.5.3 Monitores.

Los monitores se deben a Hoare (1974) y a Brinch Hansen (1977), citados en [14], y proporcionan el control tanto de la concurrencia como de la naturaleza de las operaciones sobre los datos compartidos. Un monitor es un tipo abstracto de dato (módulo) que encapsula datos, funciones privadas para manipular los datos, y una interfaz pública, que incluye las funciones y las declaraciones de tipo que se usan para manipular la información del módulo. Sólo un P/S a la vez puede estar usando una función o un tipo miembro público cualquiera del monitor. Un P/S, que solicita usar una de las funciones o tipos del monitor, es forzado a esperar por el monitor mientras haya otro P/S usando concurrentemente alguno de los miembros del monitor. Un P/S no sólo debe llamar a una función miembro para manipular el dato, sino que la ejecución de la función miembro se trata como una sección crítica.

Conceptualmente, un monitor incorpora secciones críticas en una plantilla de tipo abstracto de dato tal como se muestra en la figura 3.15.a, donde se usa el semáforo *mutex* para garantizar que sólo hay un subproceso en el monitor en cada momento.

```
TAD Monitor
{
  private:
    semaforo mutex = 1;
    <estructura de datos del TAD>
     . . .
  public:
    funcion_i(. . .)
    {
      P(mutex);
      <procesamiento de funcion_i>
      V(mutex);
    }
    . . .
}
```

a. Definición

```
Monitor SaldoCompartido
{
  private:
    int saldo;
  public:
    Depositar (int deposito)
    {
      saldo = saldo + deposito;
    }
    Retirar (int retiro)
    {
      saldo = saldo - retiro;
    }
}
```

b. Monitor de una variable compartida

Figura 3.15 Monitor

En la figura 15.b se muestra el monitor de la variable *saldo* de una cuenta de ahorros que proporciona las funciones Depositar () y Retirar ( ) para cambiar el valor,

pero protegiendo el acceso a la variable compartida con una sección crítica. Algunos procesos/subprocesos incrementarán la variable *saldo*, y otros desearán decrementarla. Aunque las instrucciones de cada función del monitor sean una secuencia de código máquina cuando se compilen, se garantiza que un subproceso será capaz de completar la secuencia completa de sentencias como una sección crítica, puesto que las sentencias aparecen dentro de una función del monitor SaldoCompartido.

Si un P/S que se ejecuta en un monitor no puede seguir adelante hasta que otro P/S cambie el estado de la información protegida por el monitor, entonces el primer P/S debe ceder temporalmente el monitor. Las variables de condición permiten esta cesión temporal y la posterior recuperación de los monitores.

Una **variable de condición**, es una estructura de datos que sólo puede aparecer dentro de un monitor, es global a sus procedimientos y su valor puede ser modificado desde tres operaciones:

- wait( ), suspende el P/S invocador sobre una variable de condición y libera el monitor hasta que otro P/S realice una señalización, signal( ).
- signal( ), reanuda un P/S actualmente suspendido sobre una variable de condición por la operación wait( ). Si no hay ningún P/S esperando señalización, signal( ) no tiene ningún efecto.
- queue( ), devuelve TRUE si hay al menos un P/S suspendido sobre la variable de condición, y FALSE en caso contrario.

La variable de condición se parece al evento visto en el párrafo anterior y tiene el mismo fin, pero en el contexto de un monitor.

Hay dos variantes del comportamiento de la operación signal( ). En la semántica de monitor de Hoare, si P1 está esperando la señal en el momento en que P0 la ejecuta desde el monitor, P0 se suspende y P1 comienza inmediatamente su ejecución en el monitor. Cuando P1 finaliza de ejecutar el monitor, P0 continúa su ejecución en el monitor. Una condición es cierta en el instante en que ocurre la señal, pero puede no serlo más tarde. En la semántica de monitor de Brinch Hansen, cuando P0 ejecuta una señal, la condición se guarda mientras P0 continúa su ejecución. Cuando P0 deja el monitor, P1 intenta continuar su ejecución en el monitor recomprobando la condición. La situación puede haber cambiado entre el instante en que Po realizó la señal y el instante en que P1 recibe el procesador. La semántica de Brinch Hansen requiere menos

cambios de contexto que la aproximación de Hoare. En la figura 3.16 el monitor de Hoare implementa un semáforo.

```
Monitor monPV
{
  private:
    bool ocupado = FALSE;
    condicion libre;
  public:
    mP( )
    {
      if(ocupado) libre.wait();
      ocupado = TRUE;
    }
    mV( )
    {
      ocupado = FALSE;
      libre.signal;
    }
}
```

Figura 3.16 Monitor implementado semáforo

En la variante de Hoare, una situación conducente a una operación "wait" es ésta:

```
. . .
if (recursoNoDiaponible) condicionRecurso.wait( );
/* Recurso ahora está disponible - continuar ... */
. . .
```

Cuando otro subproceso ejecuta una condicionRecurso.signal( ), ocurre un cambio de contexto hacia este subproceso bloqueado que recupera el monitor y continúa su ejecución en la instrucción que sigue a if. El otro subproceso señalizador se bloquea hasta que este subproceso que esperaba abandona el monitor.

En la variante de Brinch Hansen, la misma situación se puede codificar como:

```
. . .
while (recursoNoDisponible) condicionRecurso.wait();
/* Recurso ahora esta disponible - continuar ... */
. . .
```

No hay cambio de contexto hasta que el subproceso señalizador deja voluntariamente el monitor. Se garantiza que el subproceso, que ejecuta condicionRecurso.wait( ), recomprueba recursoNoDisponible antes de continuar.

3.4 COMUNICACION.

Al llevar a cabo sus funciones colectivas, los procesos/subprocesos cooperativos intercambian datos y se sincronizan unos con otros.

Los semáforos y sus abstracciones, incluyendo los monitores, están pensados principalmente para la sincronización entre procesos/subprocesos y su implementación supone en gran medida el acceso a memoria común (compartida). La aplicación directa de estos mecanismos centralizados para control de concurrencia en entornos distribuidos suele ser ineficaz y lenta.

La comunicación entre procesos/subprocesos (IPHC, inter-process/thread communication) constituye un conjunto de mecanismos para intercambiar datos entre subprocesos en uno o más procesos. Los procesos pueden estar ejecutándose en uno o más computadores conectados en red. La IPHC también puede ser referida como comunicación entre subprocesos (inter-thread communication) y comunicación entre aplicaciones (inter-application communication). La comunicación entre subprocesos de un proceso (comunicación intra-proceso) usa el mismo espacio de direcciones. La comunicación entre subprocesos de diferentes procesos (IPC) usa diferentes espacios de direcciones.

En IPC, el SO copia explícitamente la información desde el espacio de direcciones del proceso emisor a un espacio de direcciones distinto del proceso receptor. Si los dos procesos están implementados sobre la misma máquina, entonces el SO puede realizar la operación de copia soslayando el mecanismo de seguridad de la memoria. Si los procesos emisor y receptor están en máquinas diferentes, entonces el SO tendrá trabajo adicional:

- El SO del computador emisor copia la información desde la memoria del proceso emisor hacia un dispositivo de comunicación. El dispositivo transmitirá la información a un dispositivo de comunicación del computador receptor.
- El SO del proceso del computador receptor copiará después la información desde el dispositivo de comunicación hacia la memoria del proceso receptor.

Puesto que para soportar la ejecución de procesos/subprocesos concurrentes cooperativos es necesaria tanto la sincronización como la comunicación entre ellos,

sería deseable integrar ambas funciones dentro de un solo mecanismo. El mecanismo de mensajes.

3.4.1 MENSAJES.

Los mensajes constituyen un mecanismo relativamente sencillo y adecuado tanto para comunicación como para sincronización entre procesos/subprocesos en entornos centralizados además de en entornos distribuidos. El envío y recepción de mensajes es una forma estándar de comunicación entre nodos en redes de computadores, lo que hace que sea muy atractivo aumentar esta facilidad para proporcionar las funciones de comunicación y sincronización entre procesos/subprocesos.

Un mensaje es una colección de información que se intercambia entre un P/S emisor y un receptor. Un mensaje puede contener datos, órdenes de ejecución o incluso código a transmitir entre procesos/subprocesos. Los mensajes suelen ser utilizados en sistemas distribuidos para transferir porciones del SO y/o aplicaciones a nodos remotos.

El formato del mensaje es flexible y negociable por cada pareja específica emisor-receptor. Un mensaje se caracteriza por su tipo, longitud, identificadores de emisor y receptor y un campo de datos. Un posible formato de mensaje es el que aparece en la figura 3.17, que concibe el contenido del mensaje en dos campos: la cabecera y el cuerpo. La cabecera del mensaje tiene generalmente un formato fijo para cada SO. El cuerpo de mensaje, opcional, contiene el verdadero mensaje y su longitud puede variar de un mensaje a otro, incluso dentro de un mismo SO.

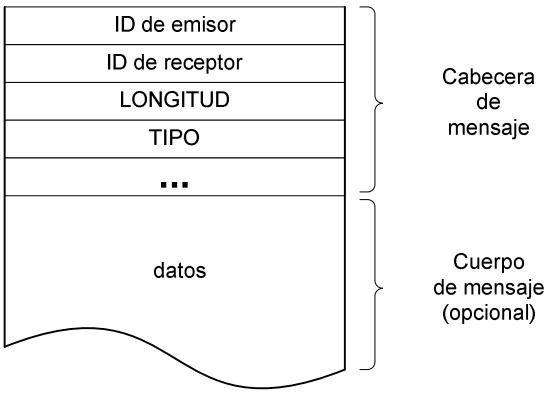


Figura 3.17 Formato de mensaje

Las operaciones típicas para soportar mensaje son enviar (send) mensaje y recibir (receive) mensaje. Las implementaciones de los mensajes difieren en una serie

de detalles que pueden agruparse en las siguientes cuestiones: denominación, copia, intercambio síncrono frente a asíncrono y longitud

La **denominación** puede ser directa o indirecta. En la denominación directa, cada emisor de mensaje debe designar al receptor específico, y, a la inversa, cada receptor debe designar a la fuente del mensaje que desea recibir; como se muestra en el ejemplo de la figura 3.18.a, donde se envía un mensaje desde el P/S A al P/S B.

```
procesoHilo A
{
    . . .
    enviar(B, mensaje);
    . . .
}
procesoHilo B
{
    . . .
    recibir(A, mensaje);
    . . .
}
```

a. Directa

```
procesoHilo A
{
    . . .
    enviar(Buzon1, mensaje);
    . . .
}
procesoHilo B
{
    . . .
    recibir(Buzon1, mensaje);
    . . .
}
```

b. Indirecta

Figura 3.18 Denominación en mensajes

En la comunicación indirecta, los mensajes son enviados y recibidos a través de depósitos especializados dedicados a ese fin, conocidos como buzones. En el ejemplo de la figura 3.18.b, el P/S A envía un mensaje a través del Buzon1 al P/S B. Esta forma de comunicación requiere servicios de mantenimiento de buzones. La comunicación indirecta puede proporcionar correspondencias de uno a uno, de uno a muchos, de muchos a uno y de muchos a muchos entre emisores y receptores.

La transferencia de mensajes del espacio de direcciones del P/S emisor al espacio de direcciones del receptor puede ser por **copia de mensaje** (valor) o por **referencia de mensaje**. En sistemas distribuidos que no disponen de memoria común, la copia es inevitable. Cuando el SO copia el mensaje desde el espacio del emisor al del receptor, ambos procesos/subprocesos permanecen desacoplados el uno del otro. La copia de mensajes consume memoria y ciclos del procesador. La transferencia de un puntero al mensaje entre los procesos/subprocesos es una solución más rápida, pero menos segura. Una consecuencia de tener una copia del mensaje es que, mientras el P/S receptor esté utilizando el mensaje, el emisor no debe modificarlo. Esto es difícil de forzar sin algún tipo de mecanismo adicional. Una estrategia híbrida es la **copia en caso de escritura**, implementada por algunos sistemas operativos.

El **intercambio de mensajes puede ser síncrono o asíncrono**. En el intercambio síncrono, el emisor y el receptor proceden juntos a completar la transferencia del mensaje. En sistemas síncronos, la operación ENVIAR es bloqueante, bloquea al emisor hasta que se emita la operación RECIBIR correspondiente. El mecanismo síncrono de enviar-recibir mensajes tiene bajo recargo, es fácil de implementar y se caracteriza porque el emisor sabe que su mensaje ha sido recibido al dejar atrás la instrucción ENVIAR. La obligada operación síncrona de emisores y receptores puede no ser deseable en algunos casos, como en los procesos/subprocesos servidores públicos.

En el intercambio asíncrono de mensajes, el emisor continúa su ejecución después de enviar un mensaje y no necesita quedar suspendido, independientemente de la actividad de los receptores. El ENVIAR asíncrono, sin bloqueo, se implementa haciendo que el SO acepte y almacene temporalmente los mensajes pendientes hasta que se emita el correspondiente RECIBIR. Por ejemplo, un P/S que desee imprimir puede incluir simplemente los datos en un mensaje y enviarlo al P/S servidor de impresora. Incluso si el servidor de impresora está ocupado en ese momento, el mensaje será almacenado en cola por el sistema, y el emisor podrá continuar sin necesidad de esperar. El almacenamiento temporal de mensajes pendientes impone un recargo adicional.

El aplazamiento indefinido ocurre cuando se envía un mensaje pero nunca se recibe, debido, por ejemplo, a la quiebra del receptor, o a que el receptor espera un mensaje que nunca se produce. Este problema se trata con la versión de RECIBIR sin espera (no bloqueante) o con la versión de RECIBIR con espera temporizada. RECIBIR sin bloqueo inspecciona el buzón indicado y devuelve el control sin suspender al invocador, entregando el mensaje encontrado o indicando el resultado.

La aproximación más completa y compleja al problema del aplazamiento indefinido es facilitar la indicación de un límite de tiempo para completar un intercambio particular de mensajes. Aunque este límite podría incorporarse en ambas operaciones, es suficiente implementarlo solo en la operación RECIBIR. Esta aproximación requiere modificar la secuencia de invocación de las operaciones de mensaje. Un ejemplo de tal secuencia se muestra en la figura 3.19.

Los mensajes pueden tener **longitud fija o variable**. El compromiso es de recargo frente a flexibilidad. La longitud de los mensajes es importante cuando los mensajes son copiados y almacenados temporalmente

```
Emisor
{
    . . .
  enviar(Buzon, mensaje);
  recibir(BuzonRecon, Recon,LimiteTiempo);
    . . .
}
```

```
Receptor
{
    . . .
  recibir(Buzon, mensaje, limiteTiempo);
  if(mensajeRecibidoATiempo)
        enviar(BuzonRecon,Recon):
    . . .
}
```

Figura 3.19 Intercambio de mensajes con espera temporizada

Los mensajes de tamaño fijo producen bajo recargo, ya que los búferes del sistema son también de tamaño fijo, resultando en una asignación sencilla y eficaz. Este tamaño fijo no es apropiado para mensajes de comunicación, los cuales suelen tener tamaños variados. Los mensajes de tamaño variable alivian este problema al crear búferes dinámicamente para que se ajusten al tamaño de cada mensaje individual, lo que produce un recargo significativo del SO y del procesador.

3.4.2 COMUNICACIÓN Y SINCRONIZACION POR MENSAJES.

Se asume un sistema de mensajes con búferes, con capacidad de canal suficiente y con buzones, que permita la operación asíncrona de emisores y receptores; en el cuál las operaciones ENVIAR y RECIBIR son indivisibles.

El intercambio de un mensaje vacío, sin datos, entre el emisor y el receptor es equivalente a la señalización entre procesos/subprocesos. El emisor ejecuta la operación V y el receptor ejecuta la operación P sobre el mismo semáforo (buzón). El P/S receptor que emite una operación P permanece suspendido hasta que el semáforo es señalizado (enviado un mensaje al buzón) por algún otro P/S. Alternativamente, si el semáforo está libre (el mensaje ya está en el buzón), el P/S que emite P recibe permiso para proseguir sin ser suspendido.

Dada la capacidad de señalización de los mensajes, la exclusión mutua puede lograrse siguiendo la misma lógica de la solución con semáforos tal como se presenta en la figura 3.20.a. El recurso compartido está protegido por el buzón *mutex*. El buzón se crea vacío. Para qué el recurso esté disponible para el primer solicitante, el P/S padre envía un mensaje al buzón antes de iniciar los procesos/subprocesos usuarios. El

contenido del mensaje es irrelevante, puede ser vacío (*nulo*). Este esquema es conceptualmente similar al de los semáforos. Un solo mensaje circula a lo largo del sistema como testigo del permiso de utilización del recurso compartido.

El programa de la figura 3.20.a muestra la implementación de semáforos binarios con mensajes. La semáforos generales pueden simularse aumentado el número de mensajes testigo.

El programa de la figura 3.20.b muestra la capacidad completa de los mensajes para transferir datos y señales simultáneamente en la solución del problema de los productores/consumidores con búfer limitado. El búfer global al que acceden todos los procesos/subprocesos ya no existe, se ha reemplazado por los mensajes. En cada mensaje hay un campo de datos para contener un dato producido.

```
Programa mjeMutex
{
  mensaje nulo;
    . . .
  procesoHiloX( )
  {
    mensaje mje;
    while(TRUE)
    {
      recibir(mutex, mje);
      <sección crítica X>
      enviar(mutex, mje);
      <otras operaciones X>
    }
  }
    . . . //otros procesos/hilos
//Ingreso mjeMutex
  {
    crearBuzon(mutex);
    enviar(mutex, nulo);
    iniciar procesoHiloX, otros procesos/hilos;
  }
}
```

a. Exclusión mutua

```
Programa mjeProdsConss
{
  mensaje nulo;
  entero capacidad; //de almacenamiento
  entero i;

  phProductorX( )
  {
    mensaje mjeP;
    while(TRUE)
    {
      recibir(puedeProducir, mjeP);
      mjeP = producir();
      enviar(puedeConsumir, mjeP);
      <otras operaciones X>
    }
  }

  phConsumindorY( )
  {
    mensaje mjeC;
    while(TRUE)
    {
      recibir(puedeConsumir, mjeC);
      consumir(mjeC);
      enviar(puedeProducir, mjeC);
      <otras operaciones Y>
    }
  }

//Ingreso mjeProdsConss
  {
    crearBuzon(puedeProducir);
    crearBuzon(puedeConsumir);
    for(i=0; i<capacidad;i++)
              enviar(puedeProducir, nulo);
    iniciar productores, consumidores;
  }
}
```

b. Productores/consumidores con búfer limitado

Figura 3.20 Sincronización y comunicación con mensajes

Los buzones, *puedeProducir* y *puedeConsumir*, se utilizan para intercambiar mensajes entre productores y consumidores. La capacidad de almacenamiento del búfer está limitada por el número total de mensajes puesto a disposición de productores y consumidores por el P/S padre al enviar inicialmente *capacidad* mensajes *nulos* al buzón *puedeProducir*.

La estrategia de mensajes aborda el problema de la manipulación global de datos, que es esencial para las soluciones vistas anteriormente, por su eliminación completa. Aparte de los buzones, todas las variables son locales. La eliminación del búfer global hace que esta solución sea aplicable tanto a entornos de procesamiento centralizado como distribuido.

Los procesos/subprocesos productores y consumidores pueden residir y ejecutarse en diferentes computadores. Por ejemplo, un computador puede albergar los productores y el buzón *puedeProducir*, y otro puede contener los consumidores y el buzón *puedeConsumir*. Siempre que los nombres de ambos buzones sean globalmente conocidos y alcanzables, y los sistemas operativos de cada instalación soporten una forma compatible de operaciones ENVIAR y RECIBIR, es posible incluso tener computadores que operen bajo diferentes sistemas operativos. Los productores y consumidores pueden ser independientemente construidos en diferentes entornos de desarrollo con diferentes lenguajes de programación.

3.4.3 SEÑALIZACION DE INTERRUPCIONES POR MENSAJES.

Las interrupciones señalan la llegada de sucesos externos. Esto es lo mismo que si un P/S señaliza un cierto suceso, tal como el final de la utilización de un recurso o el vaciado de una posición de un búfer. Esta situación plantea un mecanismo de señalización uniforme que englobe tanto a las interrupciones como a la señalización ordinaria entre procesos/subprocesos. Este mecanismo uniforme es el mecanismo de mensajes.

El SO convierte cada interrupción en un mensaje enviado por el sistema a un P/S que espera en un buzón. Por ejemplo, puede crearse un buzón dedicado por cada interrupción. Los procesos/subprocesos de servicio de interrupción pueden entonces conectarse a los sucesos externos ejecutando una operación RECIBIR en el buzón asociado cada vez que estén preparados para dar servicio a una interrupción. La llegada

de un suceso externo, indicada por una interrupción, puede ser utilizada para activar el servicio: mediante la operación ENVIAR ejecutada por el SO sobre el buzón asociado.

Esta estrategia proporciona una abstracción que delega los detalles de las operaciones de interrupción de bajo nivel al SO, y que elimina las diferencias entre las interrupciones y los procesos/subprocesos ordinarios a nivel de usuario. El servicio efectivo de una interrupción sigue descansando en el P/S servidor dedicado, proporcionando así una flexibilidad completa y haciendo innecesario que el SO se preocupe de las peculiaridades de dispositivo de las diferentes fuentes de interrupción.

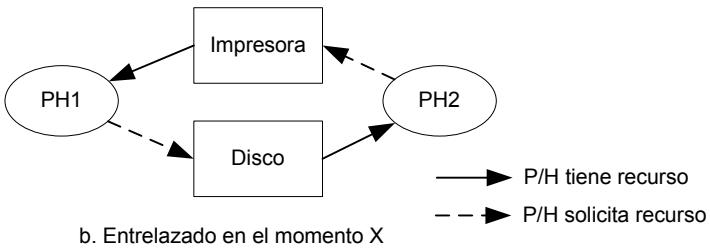
3.5 INTERBLOQUEOS.

La concurrencia y la elevada utilización de recursos proporcionan las condiciones necesarias y la base para que se produzcan interbloqueos. Un interbloqueo (deadlock) es una situación donde un grupo de procesos/subprocesos están permanentemente bloqueados como consecuencia de que cada P/S ha adquirido un subconjunto de los recursos necesarios para su operación y está esperando la liberación de los restantes recursos mantenidos por otros del mismo grupo, haciendo así imposible que ninguno de los procesos/subprocesos pueda continuar.

```
PH1
{
   . . .
  P(impresora); //solicita
//momento X: tengo impresora
   . . .
  P(disco);       //solicita
  <procesarConImpr&Disco>
  V(disco);       //libera
  V(impresora); //libera
   . . .
}
```

```
PH2
{
   . . .
  P(disco);       //solicita
//momento X: tengo disco
   . . .
  P(impresora);  //solicita
  <procesarConDisco&Impr>
  V(impresora);  //libera
  V(disco);       //libera
   . . .
}
```

a. Código de procesos/hilos



b. Entrelazado en el momento X

Figura 3.21 Interbloqueo entre dos procesos/hilos

Los interbloqueos pueden ocurrir como consecuencia de asignación incontrolada de recursos (reutilizables o consumibles) del sistema a los procesos/subprocesos

solicitantes. En la figura 3.21 se muestra un ejemplo de interbloqueo de dos procesos/subprocesos entrelazados en circunstancias específicas: PH1 tiene la impresora y solicita el disco asignado a PH2, recíprocamente PH2 tiene el disco y solicita la impresora asignada a PH1, como consecuencia ningún P/S puede continuar.

Cada P/S siempre podría completarse correctamente si ambos se ejecutasen secuencialmente, uno después de otro, no importa en que orden. El interbloqueo surge como consecuencia de la ejecución concurrente, cuando se dan simultáneamente las siguientes condiciones:

- **Exclusión mutua**: Los recursos compartidos son adquiridos y utilizados de modo mutuamente exclusivo, un P/S a la vez.
- **Poseer y esperar**: Cada P/S retiene los recursos que ya le han sido asignados mientras espera adquirir el resto.
- **No expropiación**: Los recursos concedidos a un P/S solo pueden ser liberados y devueltos al sistema como consecuencia de la acción voluntaria de ese P/S; el sistema no puede obligarle a entregarlos.
- **Espera circular**: Los procesos interbloqueados forman una cadena circular de modo que cada P/S retiene uno o más de los recursos que son solicitados por el siguiente P/S de la cadena.

Estas condiciones son necesarias para la existencia de un interbloqueo, pero su presencia no es suficiente para que se produzca un interbloqueo. La mayoría de las técnicas de los sistemas operativos para manejo de interbloqueos caen en una de las siguientes estrategias que se ven a continuación: prevención de interbloqueos, evitación de interbloqueos y detección y recuperación de interbloqueos.

3.5.1 PREVENCION.

La prevención de interbloqueos niega, por diseño del SO, al menos una de las condiciones necesarias para que se produzcan los interbloqueos, a excepción de la exclusión mutua que es generalmente difícil de evitar.

La condición poseer y esperar se elimina forzando a un P/S a liberar los recursos retenidos por él cada vez que solicite un recurso que no esté disponible. Hay dos implementaciones posibles de esta estrategia: el proceso solicita todos los recursos necesarios antes de comenzar a ejecutarse y el subproceso solicita los recursos de forma

incremental en el curso de la ejecución pero libera los recursos retenidos si se encuentra con una negativa.

La condición de no expropiación puede ser negada permitiendo expropiación, es decir, autorizando al sistema a revocar la propiedad de ciertos recursos a los procesos/subprocesos bloqueados. El SO debe encargarse de salvar el estado y restaurarlo cuando el P/S sea posteriormente reanudado. Esto es factible para recursos como el procesador y la memoria.

La condición de espera circular puede prevenirse mediante la ordenación lineal ascendente de los tipos de recursos del sistema y su asignación en ese orden a los procesos solicitantes. Los recursos del sistema se dividen en clases Cj, donde j = 1,,n. Los procesos/subprocesos solicitan y adquieren sus recursos en orden estrictamente creciente de las clases especificadas de recursos. La adquisición de todos los recursos de una clase se efectúa en una sola petición, y no incrementalmente. Una vez que un P/S adquiere un recurso de la clase Cj, solamente puede solicitar recursos de la clase j + 1 o superior. La ordenación lineal de las clases de recursos elimina la posibilidad de espera circular, ya que un P/S que tiene un recurso de la clase Ci no podrá esperar a ningún P/S que esté esperando a un recurso de la clase Ci o inferior.

3.5.2 EVITACION.

La evitación de interbloqueos es manejada por el asignador de recursos que concede recursos disponibles, solamente cuando la concesión no dar lugar a interbloqueo. Esta estrategia requiere que los procesos especifiquen (declaren) sus necesidades máximas de recursos antes de su ejecución y que cada P/S, durante su ejecución, solicite sus recursos que necesita hasta el límite declarado.

El asignador de recursos lleva la cuenta del número de recursos asignados y del número de recursos disponibles de cada tipo, además de conocer los recursos declarados no asignados, por proceso. Un P/S que solicita un recurso temporalmente no disponible espera. Si el recurso solicitado está disponible, el asignador de recursos evalúa su asignación comprobando si cada P/S activo puede completarse sin problemas, reclamando sus recursos declarados restantes. Si la evaluación es positiva, el estado del sistema después de la asignación contemplada es seguro, y el recurso es asignado al P/S solicitante. Alternativamente, el asignador de recursos suspende al P/S solicitante hasta

que el recurso solicitado puede ser concedido con seguridad, ya que la concesión del recurso podría conducir a un estado de interbloqueo,

3.5.3 DETECCION Y RECUPERACION.

En vez de sacrificar el rendimiento previniendo o evitando interbloqueos, algunos sistemas operativos conceden libremente los recursos disponibles a los procesos/subprocesos solicitantes, y comprueban ocasionalmente el sistema para determinar si existen interbloqueos con el fin de reclamar los recursos retenidos por los procesos/subprocesos interbloqueados.

Conocidos algoritmos son usados para detectar los circuitos o nudos en el grafo general de recursos del sistema que acreditan la presencia de interbloqueos y los procesos/subprocesos implicados. El grafo general de recursos es una representación de los procesos/subprocesos y recursos del sistema y de las relaciones entre ellos. La figura 3.21.b es un ejemplo de este tipo de grafo.

La frecuencia de invocación del algoritmo es un parámetro de diseño del SO. Una posibilidad es comprobar los interbloqueos cada vez que se solicita un recurso. Otra alternativa es activar el algoritmo de detección de interbloqueos ocasionalmente, a intervalos regulares o cuando uno o más procesos/subprocesos queden bloqueados durante un largo tiempo.

El primer paso en la recuperación de interbloqueos es identificar los procesos/subprocesos interbloqueados. Los algoritmos de detección proporcionan una indicación de los procesos/subprocesos interbloqueados. El siguiente paso es romper el interbloqueo volviendo atrás o reiniciando desde el principio uno o más de los procesos/subprocesos interbloqueados. En la reiniciación es deseable elegir las víctimas entre los procesos/subprocesos que hayan llegado menos lejos y que su recuperación sea menos onerosa. La vuelta atrás requiere una utilidad que registre los estados en tiempo de ejecución de los procesos/subprocesos, de modo que puedan regresar a un instante suficientemente anterior en el pasado para que el interbloqueo se rompa. En general, tanto la vuelta atrás como la reiniciación pueden ser difíciles, si no imposibles, para procesos que han efectuado cambios irreversibles sobre los recursos de que disponían antes de que se produjera el interbloqueo.

Capitulo IV:
GESTOR ACADEMICO DE SUBPROCESOS.

Como se describió en los dos capítulos anteriores, la gestión de procesos implementa los procesos, los subprocesos y la abstracción de recursos (excepto memoria, dispositivos y archivos) en el hardware del computador, controlando la actividad del procesador y de los demás recursos para llevar acabo la ejecución de programas. El proceso presenta dos dimensiones: la relacionada con la posesión de recursos (memoria, dispositivos de entrada/salida, archivos) y la relacionada con la ejecución - motor de ejecución (procesador). Estas dos características esenciales del proceso pueden ser tratadas de forma independiente por el SO. La mayoría de sistemas operativos actuales destacan motores de ejecución en su propia construcción con nombres tales como: subproceso (sP), hilo, hebra, proceso ligero u otro. El procesador se asigna a los subprocesos. En el proceso hardware de los computadores personales actuales, el procesador realiza hasta cientos de millones de instrucciones de lenguaje máquina por segundo ejecutando subprocesos (sPP) dentro de procesos del sistema operativo y de programas de aplicación.

El proceso de enseñanza – aprendizaje de subprocesos y procesos en el SO, particularmente por el lado demostrativo y experimental, requiere de programas que describan los procesos y subprocesos en un lenguaje de programación general y de herramientas que permitan su ejecución referencial a este lenguaje de programación a velocidades manejables. La ejecución referencial permite emular la ejecución directa de las instrucciones del lenguaje de programación general. La velocidad de ejecución referencial esta asociada a la velocidad del proceso de enseñanza – aprendizaje de los interesados.

La gestión de subprocesos para fines académicos que se propone se refiere fundamentalmente a los motores de ejecución y se materializa en el Gestor Académico de Subprocesos (GASP) como una herramienta para el proceso de enseñanza – aprendizaje de subprocesos y procesos, que gestiona subprocesos (motores de ejecución) en la ejecución de programas, que emplea un lenguaje de programación general OO para describirse y describir a los subprocesos y que permite controlar la velocidad de ejecución referencial y las características importantes de los subprocesos

por el interesado. El GASP es un marco de trabajo en el cual se gestionan y se ejecutan los subprocesos. El GASP brinda los servicios de núcleo del SO, especificado en el párrafo 2.2.4, para la gestión y ejecución de subprocesos; así como también brinda una interfaz al usuario para la configuración y conducción del mismo y de los subprocesos.

Este capítulo define el GASP por medio de sus servicios, el factor académico que incorpora, los aspectos básicos de su interfaz y finalmente por su especificación funcional.

4.1 DEFINICION DE SERVICIOS DEL GASP.

El GASP es un emulador del núcleo de un sistema operativo multitarea (múltiples motores de ejecución) basado en memoria que combina aspectos importantes vistos en capítulos anteriores, tales como las interrupciones, la administración de sPP, las interacciones entre sPP y la planificación de sPP. Los mensajes son sus mecanismos de sincronización y comunicación y emplea la planificación expropiativa de sPP guiada por sucesos para la planificación básica.

Los mensajes brindan un conjunto de primitivas funcionalmente completo para la comunicación y la sincronización entre sPP y, además, son utilizados para encaminar las interrupciones desde las fuentes hardware a los sPP de servicio de interrupción. La sincronización incluye la señalización y la exclusión mutua.

La planificación guiada por sucesos, es decir, la planificación basada en prioridades es usada por el GASP para efectuar la planificación básica con el fin de controlar la ejecución de los sPP.

Los servicios proporcionados por el GASP para la gestión y ejecución de sPP se definen en servicios y las responsabilidades internas que los viabilizan se presentan en el soporte de servicios. En base a estos servicios y al soporte de servicios se construye el diagrama de transiciones de estado del GASP, presentado en transiciones de estado de los subprocesos.

4.1.1 SERVICIOS.

La funcionalidad proporcionada por el GASP a sus usuarios se expresa en los siguientes servicios, organizados por funciones generales:

Sincronización y comunicación entre sPP. Los mensajes son los mecanismos para estos servicios. Para el uso de mensajes se emplean las operaciones ENVIAR MENSAJE y RECIBIR MENSAJE. Para lograr una correspondencia general de muchos a muchos entre sPP emisores y sPP receptores, GASP soporta la gestión de buzones.

Gestión de interrupciones. Puesto que se emplean mensajes para señalizar la ocurrencia de interrupciones hardware a los sPP encargados de su procesamiento, es suficiente proporcionar servicios para manipular los niveles de interrupción, tales como la habilitación de un nivel. La operación HABILITAR NIVEL INTERRUPCION es proporcionada por GASP para habilitar niveles interrupción hardware.

Administración de sPP. CREAR SUBPROCESO es la primera operación. La suspensión y reanudación de subprocesos se consigue indirectamente por medio de las operaciones de mensajes, por lo que GASP no proporciona estas operaciones. Para terminar subprocesos se proporcionan las operaciones: FINALIZAR SUBPROCESO y ABORTAR SUBPROCESO, la primera se invoca por un sP cuando finaliza ordenadamente y la segunda es llamada por el usuario para terminar (abortar) en cualquier momento un sP. Con el fin de soportar planificación guiada por tiempo (reloj), GASP proporciona la operación DORMIR SUBPROCESO para diferir la ejecución de un subproceso por un determinado tiempo.

Configuración inicial. GASP está orientado a ambientes ms Windows con plataforma .NET en computadores personales, a la memoria de los cuales se carga junto con el código de los sPP como un programa ejecutable. Después de la carga, GASP define su configuración inicial de su ambiente hardware/software, quedando listo para ser habilitado y usado por el usuario final.

4.1.2 SOPORTE DE SERVICIOS.

Para soportar los servicios proporcionados, el GASP desempeña responsabilidades que incluyen: procesamiento de mensajes, procesamiento de interrupciones, gestión de base de tiempos, planificación y despacho de sPP, configuración de hardware y su propia configuración.

Los mensajes se emplean para el acceso exclusivo a recursos por los sPP, para la señalización entre sPP y para señalizar la ocurrencia de interrupciones hardware. Estos mensajes son de denominación indirecta por medio de buzones, de transferencia por

referencia, de almacenamiento en búferes (buzón o descriptor de sP), de intercambio asíncrono y de longitud variable.

Los mensajes señalan la llegada de las interrupciones hardware a los sPP dedicados a su procesamiento. Esta característica se obtiene dedicando un buzón a cada fuente potencial de interrupción hardware, como por ejemplo temporizador de intervalo programable, teclado y otros dispositivos. Cada vez que se produce una interrupción (de nivel mayor que la del subproceso en ejecución) por parte de un dispositivo, GASP efectúa el procesamiento inicial de bajo nivel y envía un mensaje de interrupción al buzón afectado. Un sP que da servicio a interrupciones procedentes de una determinada fuente recibe los correspondientes mensajes mediante llamadas a RECIBIR MENSAJE dirigidas al buzón dedicado. Obviamente, GASP también efectúa la gestión del nivel de interrupciones por enmascaramiento de acuerdo a la prioridad del sP en ejecución, conmutación de contexto (si es necesario) y algún tipo de reconocimiento de interrupción.

La base de tiempos del GASP genera su reloj y su unidad de tiempo. El periodo del reloj se determina como un elemento del factor académico y se define en un párrafo posterior de este capítulo. La unidad de tiempo es un múltiplo del reloj y es la unidad de medida del tiempo que puede dormir un sP que solicita este estado. Las unidades de tiempo son suministradas como parámetro a la operación DORMIR SUBPROCESO.

Al cargarse en memoria, el GASP configura la base de tiempos, emula el controlador de interrupciones programable y determinados registros del procesador, prepara el mecanismo de interrupciones (buzones, mensajes y objetos – interrupción), crea la configuración inicial de subprocesos y, finalmente, muestra su interfaz grafica de usuario y espera. Al habilitarse la base de tiempos por el usuario, el control pasa al sP listo de más alta prioridad. Siempre existe, por lo menos, un sP listo dispuesto a ejecutarse en el GASP. Desde este momento el GASP actúa guiado por sucesos en el sentido de que los patrones de ejecución de los sPP, junto con la ocurrencia de sucesos que ocasionan replanificación, dictan las transiciones de estado posteriores.

El GASP realiza una planificación sencilla de sPP listos basada en prioridad, y despacha para su ejecución el sP listo de más alta prioridad.

El GASP lleva la cuenta de los sPP y de sus cambios de estado individuales manteniendo un descriptor de subproceso (DSP) por cada sP activo. El DSP es creado

dentro de la operación CREAR SUBPROCESO y almacena los atributos del sP, entre ellos su prioridad (estática) y su referencia.

Un sP recién creado queda listo para ejecutarse, y su DSP es puesto en la lista de d sPP listos donde espera su selección por parte del planificador y su despacho a ejecución. Después de una ráfaga de actividad del procesador, el sP en ejecución puede ceder control al GASP o ser expropiado como resultado de la ocurrencia de una interrupción. En un sistema multitarea, tal como el GASP, puede haber activa una serie de sPP en diferentes etapas de procesamiento en cada momento. El estado colectivo es registrado por medio de listas de descriptores de sPP. Por esta razón el GASP tiene diferentes listas, incluyendo la lista de sPP listos y la lista de sPP bloqueados

4.1.3 TRANSICIONES DE ESTADO DE LOS SUBPROCESOS.

Sobre la base de los servicios definidos y del soporte que realiza el GASP para proporcionarlos, se deduce el diagrama de transiciones de estado, propio del gestor académico de subprocesos que se presenta en la figura 4.1. Este diagrama muestra los estados por los cuales transita un sP durante su vida en el GASP. Los estados dormido y esperando procesos son derivados del estado bloqueado tratado en el capítulo anterior.

Los sPP se crean por medio de la llamada a CREAR SUBPROCESO. A un sP recién creado se le asigna un DSP y queda listo para ejecución, tal como indica la flecha entre los estados inactivo y listo en la figura 4.1. Cada vez que se procesa un suceso que produce replanificación, GASP planifica el sP listo de mayor prioridad y lo despacha a ejecución. El sP en ejecución realiza su flujo de instrucciones hasta ser expropiado por una interrupción de mayor prioridad o hasta ceder voluntariamente el control al GASP mediante la ejecución de uno de sus servicios. En función del estado del sistema y del servicio invocado, GASP puede o no devolver el control al sP invocante.

Como muestra la figura 4.1, el sP en ejecución vuelve a obtener el control inmediatamente después de crear un nuevo sP, de crear un buzón o de eliminar un buzón. El sP en ejecución puede terminarse a si mismo y abandonar el GASP por medio de la llamada a FINALIZAR SUBPROCESO.

El sP en ejecución se expropia enviando un mensaje a un sP de mayor prioridad que ya este esperando. Cuando el mensaje sea entregado por GASP como resultado de ejecución de la operación ENVIAR MENSAJE, los sPP emisor y receptor estarán listos,

pero se planificará el receptor por ser de mayor prioridad. Dado que el intercambio de mensajes es asíncrono, el envío de un mensaje en cualquier otro caso (aún cuando el sP que ya esté esperando sea de menor prioridad y pasa al estado listo al recibir el mensaje) hace que el sP invocante mantenga el control del procesador después de ejecutarse ENVIAR MENSAJE.

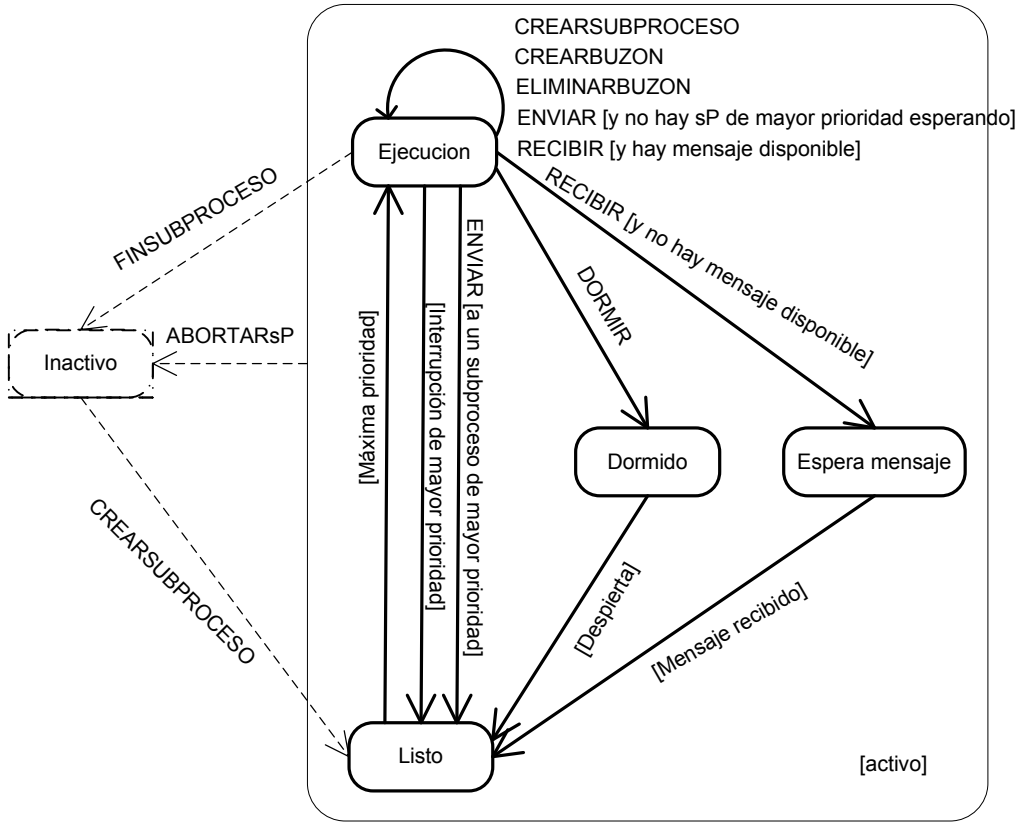


Figura 4.1 Transiciones de estado de subprocesos

La operación DORMIR SUBPROCESO suspende al sP invocante durante el tiempo especificado en unidades de tiempo del GASP. Este tiempo el sP lo pasa en el estado dormido, del cual sale al despertarse y pasa al estado listo.

Los sPP entran en el estado esperando mensaje después de invocar la operación RECIBIR MENSAJE que no encuentra mensajes disponibles en el buzón indicado. Cuando hay disponibles uno o más mensajes en el buzón designado, la operación RECIBIR MENSAJE devuelve tanto el control como el mensaje al sP invocante.

La flecha “interrupción de mayor prioridad” que va del estado en ejecución al estado listo, indica la expropiación del sP en ejecución por interrupción hardware. En

este caso se planifica y se despacha el sP que atiende la interrupción hardware u otro sP de mayor prioridad, incluyendo el recientemente creado.

Finalmente, la flecha que va desde el estado activo al esta inactivo, señala que un proceso puede ser abortado (terminación forzosa) estando en cualquiera de los estados activos.

4.2 FACTOR ACADEMICO.

Los conocimientos sobre sPP y procesos de sistemas operativos son compartidos por los interesados (docentes y estudiantes) en su enseñanza – aprendizaje, como parte del proceso enseñanza - aprendizaje presentado en el primer capítulo, a través de medios (y material), técnicas y herramientas dentro de un contexto determinado. Gran parte de este material y herramientas lo constituyen los programas operativos y aplicativos pertinentes. La comprensión de la organización de estos programas y de su dinámica de ejecución facilitan este proceso de enseñanza – aprendizaje. Relacionados con esta comprensión se encuentran los lenguajes de programación empleados en la construcción y ejecución de estos programas, así como el control de su velocidad y dinámica de ejecución en el computador por parte de los interesados; factores que se presentan en los siguientes dos párrafos.

4.2.1 LENGUAJE DE PROGRAMACIÓN.

Los programas de computadora suelen escribirse (editarse) en lenguajes de programación orientados al usuario - persona (desarrollador, programador, docente, estudiante) y se ejecutan en el lenguaje de la máquina destino.

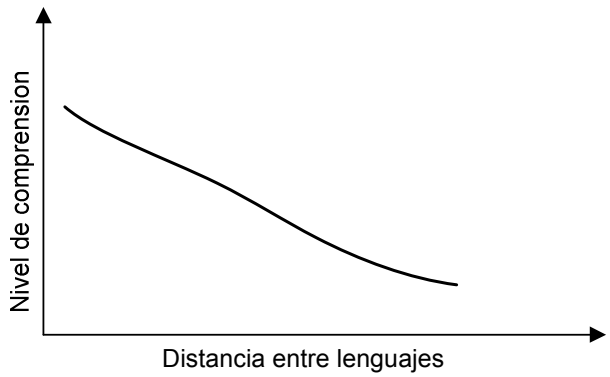


Figura 4.2 Relación entre distancia y comprensión

Normalmente existe una distancia entre el lenguaje de programación (edición) elegido y el lenguaje máquina. La relación entre esta distancia y la comprensión de los sPP y procesos del SO se presenta en la figura 4.2. La cual muestra que se logra una mayor comprensión si el lenguaje de programación coincide con el lenguaje de máquina (distancia 0).

El proceso de construcción de programas cubre esta distancia, pasando por diferentes etapas, entres las que se destacan:

- **Edición del programa en el lenguaje de programación**, el interesado ingresa su código y se genera el programa fuente. En este trabajo se ha elegido un lenguaje de programación general orientado a objetos. En este lenguaje se especifica en forma clara para el interesado: tanto el GASP como los programas aplicativos.
- **Traducción del programa fuente al lenguaje máquina**. El compilador (programa) genera el código **objeto** en el lenguaje máquina, tomando como entrada el programa fuente. Frecuentemente el código objeto es generado en un lenguaje intermedio, que es traducido al lenguaje máquina durante la ejecución; como es el caso para el ambiente de programación elegido para este trabajo. Cada instrucción en lenguaje fuente se convierte en varias instrucciones en lenguaje máquina.
- **Enlazado de programas en código objeto con facilidades operativas para crear el programa ejecutable**. El enlazador (programa) integra los módulos en código objeto con las facilidades que brinda el entorno de construcción de programas, creando el programa ejecutable. Este programa es el que se carga en la memoria del computador, a partir del cual se crea el proceso y subprocesos para su ejecución.

El interesado describe tareas en un lenguaje de programación general orientado a objetos, el computador ejecuta esas tareas en su lenguaje máquina. El seguimiento de la ejecución de las tareas en el computador, respecto a lo descrito claramente en el programa fuente, es indirecto; lo cual dificulta la comprensión del proceso y subprocesos en el sistema operativo.

El GASP permite reducir la distancia entre el lenguaje de programación y el lenguaje máquina brindando la facilidad para que se acredite la ejecución del código en lenguaje máquina correspondiente a cada instrucción o grupo apropiado de instrucciones del programa fuente, facilitando así el seguimiento de las tareas ejecutadas por el computador y descritas en el programa fuente. A esta acreditación del avance del procesamiento del programa ejecutable, mostrando la instrucción o grupo de

instrucciones correspondientes del programa fuente en tiempos apropiados para el interesado, se le denomina ejecución referencial. La acreditación se realiza en términos de instrucciones del programa fuente y/o datos, producidos por estas instrucciones. Esta acreditación se formula en interfaces gráficas, en forma visual para el interesado.

4.2.2 VELOCIDAD DE EJECUCION DE PROGRAMAS.

Un microprocesador de 3 GHz de computador personal, caracterizado por un juego de instrucciones con promedio de tiempo de ejecución de 10 periodos de reloj, procesa 300 millones de instrucciones de lenguaje máquina por segundo. Considerando 10 instrucciones de lenguaje máquina por instrucción del lenguaje de programación, un computador con este tipo de procesador ejecuta 30 millones de instrucciones de programa fuente por segundo. A esta velocidad, el interesado difícilmente captaría por algún medio (por ejemplo visual) la dinámica de ejecución de la tareas (sPP o procesos), especificadas en el programas fuente. El proceso de enseñanza – aprendizaje de los subprocesos y procesos de los sistemas operativos es afectado por la velocidad del computador al ejecutar los programas usados para esta finalidad. Esta influencia se refleja en el gráfico de la figura 4.3, donde la ejecución referencial que permite acreditarse aceptablemente ronda alrededor de una instrucción de programa fuente por segundo. Tanto la alta velocidad como la muy baja velocidad del computador parecen inadecuadas para esta finalidad.

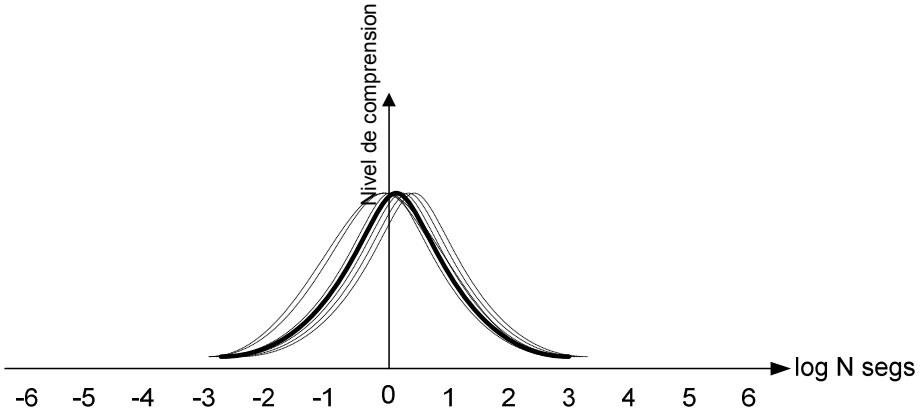


Figura 4.3 Velocidad de ejecución de programas y comprensión de su dinámica.

El GASP permite el control de la velocidad y de la dinámica de ejecución de los sPP que gestiona, facilitando al interesado ingresar la velocidad de ejecución y administrar determinados objetos tales como mensajes y buzones.

La base de tiempos del GASP se encarga de generar el reloj que define la velocidad de ejecución de los sPP a partir del pedido del interesado y de un objeto temporizador del computador que genera milisegundos.

4.3 INTERFAZ.

El GASP es un programa de escritorio (desktop) para instalarse sobre plataformas .NET de Microsoft o compatibles, que proporciona servicios a los subprocesos que gestiona por medio de una interfaz dedicada y que se comunica con el usuario final a través de una interfaz gráfica. En los siguientes párrafos de resumen estas interfaces.

4.3.1 INTERFAZ GRAFICA DE USUARIO FINAL.

El usuario final (interesado) maneja el GASP a través de la interfaz gráfica, cuyo prototipo se presenta en la figura 4.4. Esta interfaz tiene dos áreas: sistema y administrar subprocesos. Administrar subprocesos es la IGU de los subprocesos hospedados en el GASP.

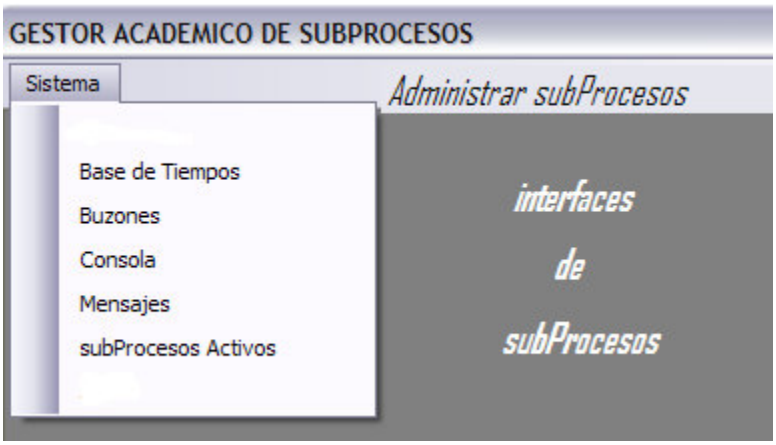


Figura 4.4 Interfaz gráfica de usuario final

En el área de sistema el interesado controla los siguientes objetos:

- Base de tiempos, para establecer el reloj y las unidades de tiempo y para habilitar/deshabilitar la base de tiempos.
- Buzones, para crear y eliminar buzones.
- Consola, para ver la evolución (transición de estados) de los subproceso activos.
- Mensajes, para crear y eliminar mensajes.
- subProcesos Activos, para mostrar los subprocesos activos y abortarlos (si se desea).

Por medio de la segunda área, el interesado crea las interfaces visuales de administración de los sPP interactivos, a través de las cuales se crean estos subprocesos y se acreditan los avances de se ejecución. Durante la creación de las interfaces de administración de sPP, se suministran sus nombres, sus prioridades y otros atributos; así como se eligen los objetos que son usados por estos subprocesos.

4.3.2 INTERFAZ CON LOS SUBPROCESOS.

La interfaz entre el GASP y los sPP está constituida por los servicios proporcionados por el GASP para gestionarlos. Estos servicios se definieron en el párrafo servicios y se caracterizan en el párrafo especificación funcional.

4.3.3 OTRAS RESTRICCIONES.

El GASP se instala sobre la plataforma .NET de Microsoft, versión 1.1 y posteriores. Esta plataforma establece los requisitos de SO y hardware para la ejecución, así como las características deseables del ambiente de programación para construir los programas. Los principales sistemas operativos corresponden a la línea de Microsoft, entre los que se encuentran Windows xp, Windows vista, Windows 7. Cualquier computador, que soporta alguno de los sistemas operativos mencionados, satisface los requisitos de hardware, entre ellos las PCs y laptops convencionales. Los programas se han codificado en el lenguaje orientado a objetos C#, empezándose en el IDE vs.net 2003 y completándose en las versiones posteriores.

Los subsistemas y/o paquetes del GASP y de las tareas que gestiona se organizan en capas. El modelamiento de estos subsistemas y/o paquetes, así como de sus contenidos, se orienta a objetos.

Finalmente y dado que el GASP se carga en memoria del computador junto con los sPP que gestiona (sistema basado en memoria), sus servicios son invocados por la modalidad de llamada a operación (procedimiento).

4.4 ESPECIFICACION FUNCIONAL.

El análisis de los requisitos funcionales y no funcionales planteados en los párrafos anteriores de este capítulo, practicado a la luz de los conceptos desarrollados en los capítulos anteriores y dentro de las restricciones asumidas, produce la especificación funcional del GASP. Esta especificación define los requisitos funcionales que debe cumplir el software que implementa el gestor académico de sPP. La especificación se

presenta en operaciones organizadas por funciones generales. Por defecto, el servicio es externo (se invoca por usuario: sP o servicio de interfaz), en caso contrario (no se invoca por usuario) se indicará explícitamente.

4.4.1 PLANIFICACION BASICA DE SUBPROCESOS

La lista de sPP listos referencia a los subprocesos listos para ejecución. La planificación básica de sPP selecciona el subproceso listo de mayor prioridad y lo despacha a ejecución. Para planificación básica, el GASP proporciona las operaciones PLANIFICAR SUBPROCESO y DESPACHAR SUBPROCESO.

Operación PLANIFICAR SUBPROCESO. Operación interna que selecciona el subproceso con mayor prioridad en la lista de sPP listos y lo referencia para ejecución (la variable global que apunta al proceso en ejecución). El control de procesador es pasado a la operación DESPACHAR SUBPROCESO.

Operación DESPACHAR SUBPROCESO. Operación interna que entrega el control del procesador a un sP para iniciar o para continuar su ejecución. Prepara la máscara para el registro de interrupción del CIP a partir de las máscaras de interrupción del sistema y de sP referenciado para ejecución. Establece el registro de interrupción del CIP con la máscara preparada. Cambia el estado del sP referenciado para ejecución de "listo" (u otro) a "En Ejecución" (solamente el sP en estado "En Ejecución" es pulsado por el reloj del GASP y se ejecuta). Pasa el control al sP referenciado para ejecución y con estado "En Ejecución".

4.4.2 SINCRONIZACION Y COMUNICACIÓN DE SUBPROCESOS.

Los mensajes con almacenamiento en búferes (buzones) son los mecanismos empleados en la comunicación y sincronización entre sPP. Se soporta la gestión de buzones y mensajes. Se proporcionan las primitivas de ENVIAR MENSAJE y RECIBIR MENSAJE.

El buzón mantiene los mensajes que esperan ser entregados o los sPP que esperan mensajes. En cualquier instante, en un buzón se da solo una de las siguientes tres posibilidades: no hay mensajes ni sPP esperando, hay mensajes esperando ser entregados o hay sPP esperando mensajes. Tanto los mensajes como los sPP que esperan en un buzón se organizan en cola (primero en llegar, primero en ser

entregado/servido). Los buzones se organizan en una lista para facilitar el cumplimiento de los servicios del GASP.

Los mensajes de comunicación contienen los datos que se transfieren, los mensajes de sincronización no precisan contener datos.

El GASP proporciona las siguientes operaciones: ENVIAR MENSAJE, RECIBIR MENSAJE y operaciones para gestión de buzones.

Operaciones de gestión de buzones. Entre las operaciones que gestionan la lista de buzones del GASP se encuentran: insertar buzón, eliminar buzón, leer buzones.

Operación ENVIAR MENSAJE. Esta operación envía mensajes al buzón, especificados como parámetros. Si hay subproceso esperando mensaje: la operación entrega el mensaje al sP receptor y lo pasa al estado listo, luego planifica y despacha el sP de mayor prioridad. En caso contrario, deposita el mensaje en el buzón destino y devuelve el control al invocador.

Operación RECIBIR MENSAJE. Esta operación entrega un mensaje al invocador, tomándolo del buzón especificado como parámetro. Si hay mensajes esperando: extrae el mensaje del buzón y lo entrega al invocador, al cual le devuelve el control. En caso contrario: suspende el sP invocante en el buzón y planifica y despacha el siguiente sP.

4.4.3 GESTION DE INTERRUPCIONES.

El servicio de interrupciones se basa en el mecanismo de intercambio de mensajes. La interrupción es interceptada por el GASP y convertida en un mensaje, el cual es entregado, por medio de un buzón dedicado, al sP específico que atiende la interrupción. Cada sP tiene su máscara de interrupciones, deducida de su prioridad, que define si puede ser interrumpido. Las prioridades más altas son asignadas a sPP que atienden interrupciones y son agrupadas en prioridades hardware o niveles de interrupción. Los dispositivos con mayor nivel de interrupción pueden interrumpir a sPP que atienden interrupciones de menor nivel y a cualquier otro sP.

El GASP emula el controlador de interrupciones programable (CIP) para el enmascaramiento selectivo de interrupciones y la máscara de interrupciones del sistema para indicar que dispositivos están habilitados para interrumpir y tienen sus correspondientes sPP activados. La máscara de interrupción de sPP, la máscara de

interrupción del sistema y la máscara de interrupción del CIP son registros de la misma longitud con contenido en lógica positiva y con asignación de bits de menor numeración a niveles de interrupción de mayor prioridad. Además, el GASP emula la habilitación y deshabilitación del sistema de interrupciones en general para proteger su código que no debe ser interrumpido.

El establecimiento de los niveles de interrupción es realizada por la operación DESPACHAR SUBPROCESO para todos los sPP, incluyendo los que atienden a interrupciones.

Para soportar la gestión de interrupciones, GASP proporciona las operaciones de SERVIR INTERRUPCIONES y HABILITAR NIVEL DE INTERRUPCION.

Operación SERVIR INTERRUPCIONES. Esta operación interna realiza un proceso inicial de las interrupciones. Las interrupciones reconocidas por el GASP y habilitadas invocan a esta operación ("invocación hardware"), indicando su nivel de interrupción (bit del registro de interrupciones del CIP). La operación expropia el sP en ejecución, identifica buzón dedicado y arma mensaje de interrupción, reconoce interrupción a dispositivo CIP y envía mensaje a buzón dedicado. Si no ha hay sP esperando, la operación planifica y despacha sP listo de mayor prioridad. En relación con esta operación, el GASP emula el temporizador de intervalo programable (TIP) como dispositivo que genera interrupciones periódicamente y le asigna el mas alto nivel de interrupción (bit 0 del registro de interrupciones del CIP).

Operación HABILITAR NIVEL DE INTERRUPCION. Esta operación habilita el nivel de interrupción (especificado como parámetro) de la mascara de interrupciones del sistema. La operación pone a uno el nivel (bit) de la máscara del sistema, indicando que está habilitada la interrupción asociada con el sP invocador, y luego devuelve el control a este sP invocador.

4.4.4 ADMINISTRACION DE SUBPROCESOS.

Cada sP activo tiene su descriptor (DSP) que nace en la operación CREAR SUBPROCESO y que desaparece en la operación FINALIZAR SUBPROCESO o en la operación ABORTAR SUBPROCESO. Empleando los descriptores, el GASP organiza listas para administrar los sPP activos: lista de todos los sPP, lista de los sPP listos, lista de los sPP dormidos y lista fraccionada de sPP en espera de mensajes. Cada fracción de

esta última lista se asocia al buzón donde los sPP esperan mensajes. Un sP ingresa a la lista de sPP dormidos por medio de la operación DORMIR SUBPROCESO. El sP DESPERTAR SUBPROCESO despierta los sPP dormidos, regresándolos a la lista de sPP listos, como resultado del procesamiento de las interrupciones del TIP.

El GASP proporciona las operaciones mencionadas y las operaciones requeridas para gestionar sus listas.

Operación CREAR SUBPROCESO. Esta operación registra un sP con sus atributos especificados como parámetros y lo deja listo para ser planificado y despachado. Inserta sP con sus atributos en lista de todos los sPP y obtiene su DSP. Inserta DSP en lista de sPP listos. Devuelve le control a invocador.

Operación FINALIZAR SUBPROCESO. Esta operación finaliza naturalmente el sP indicado por el DSP, especificado como parámetro. Retira el DSP del sP de la lista de sPP listos. Elimina el DSP de la lista de todos los sPP. Planifica y Despacha el siguiente sP.

Operación ABORTAR SUBPROCESO. Esta operación termina forzosamente un sP indicado por el nombre, especificado como parámetro. Obtiene el DSP del sP a partir de su nombre. Aborta el hilo. Elimina el DSP de las filas en las que se encontraba. Si el sP estaba en ejecución se planifica y despacha el siguiente sP, en caso contrario se devuelve el control a invocador.

Operación DORMIR SUBPROCESO. Esta operación pone a dormir al sP invocante durante el tiempo especificado como parámetros. Retira el DSP del sP. Inserta el sP en la lista de sPP dormidos, indicando el tiempo a dormir. Planifica y despacha el siguiente sP.

Servicio DESPERTAR SUBPROCESO. Este servicio se implementa como un sP que da servicio a las interrupciones del TIP, en consecuencia es invocado por interrupciones de este dispositivo (hardware). Esta interrupción tiene la máxima prioridad en el GASP. El sP habilita el bit 0 de la máscara de interrupciones del sistema y entra a su bucle sin fin. Recibe el mensaje de interrupción del TIP en el buzón asignado a este nivel de interrupción. Decrementa el tiempo de los subprocesos de la

lista sPP. Cada sP que terminó de dormir es retirado de la lista de sPP dormidos en insertado en la lista de sPP listos.

Operaciones de gestión de lista de todos los subprocesos. Entre las operaciones que gestionan la lista de todos los sPP se encuentran: insertar DSP, eliminar DSP, buscar DSP, leer DSPs.

Operaciones de gestión de lista de subprocesos listos. Entre las operaciones que gestionan la lista de los sPP listos se encuentran: insertar DSP, eliminar DSP.

Operaciones de gestión de lista de subprocesos dormidos. Entre las operaciones que gestionan la lista de los sPP dormidos se encuentran: insertar DSP, eliminar DSP.

4.4.5 CONFIGURACION INICIAL.

El GASP, su interfaz de usuario y los subprocesos que gestiona, como un programa ejecutable se cargan juntos en la memoria del computador. El GASP se instancia por su interfaz de usuario, se crean las variables que se usan y el control pasa a la operación interna CONFIGURACION INICIAL.

La operación CONFIGURACION INICIAL realiza el siguiente trabajo:

- deshabilita el sistema de interrupciones del GASP,
- se crean los objetos que emulan el hardware,
- se inicializa la mascara de interrupción del GASP,
- se inicializa el sistema de interrupciones: buzones, mensajes objetos de interrupción,
- se configuran el reloj del GASP
- se configuran los sPP que siempre deben estar activos
- se planifica un sP listo
- se habilita el sistema de interrupciones del GASP
- se despacha el sP planificado.

En el siguiente capítulo se diseña y se implementa (programa) el GASP. En el diseño se emplea un modelador UML. En la programación se trabaja con el entorno integrado de desarrollo vs.net 2005 de Microsoft.

Capitulo V:

DISEÑO E IMPLEMENTACION DEL GASP.

Los requisitos y restricciones del gestor académico de subprocesos (GASP), así como su especificación funcional, resultado del análisis de los requisitos y restricciones, fueron presentados en el capítulo anterior. Sobre la base de estos requisitos, restricciones y especificación, en este capítulo se presenta el diseño e implementación del GASP con un enfoque orientado a objetos.

El resumen y precisión de los requisitos no funcionales y restricciones más importantes del GASP, así como el planteamiento de los medios y formas de su cumplimiento se presentan en consideraciones de diseño e implementación. El diseño se presenta en: arquitectura, clases y objetos del GASP, así como en interfaz de integración del GASP. La implementación se organiza en: arquitectura de componentes y despliegue, componentes del GASP y componentes de la interfaz del GASP.

5.1 CONSIDERACIONES DE DISEÑO E IMPLEMENTACION.

El GASP es una aplicación desktop y multisubproceso, que permite al usuario gestionar subprocesos (tareas) por medio de interfaces graficas y amigables. La aplicación es instalable sobre la plataforma .NET 1.1 o posteriores de Microsoft y se ha construido en el entorno ms vs.net 2003 y posteriores con lenguaje C#. La velocidad y dinámica de ejecución de los subprocesos (sPP) son controlables por el usuario. La ejecución del subproceso (sP) es referenciada al lenguaje fuente (C#) y es acreditada amigablemente en la interfaz del sP.

Entre las consideraciones de diseño e implementación más importantes, se encuentran: elementos de hardware emulados, control de velocidad de ejecución de subprocesos, acreditación de ejecución referencial de subprocesos y conmutación de subprocesos.

5.1.1 ELEMENTOS DE HARDWARE EMULADOS.

El GASP requiere modelar los siguientes elementos – hardware del computador personal: el reloj (clock), el temporizador de intervalo programable (TIP), el controlador de interrupciones programable (CIP), algún registro del procesador relacionado con

habilitación selectiva de interrupciones (Mascara), un lugar de memoria del computador y la habilitación/deshabilitación general de interrupciones del computador.

La base de tiempos del GASP realiza el clock y el TIP. El periodo del clock es de 600 ó más milisegundos y se implementa en un temporizador del .NET con el número de milisegundos ingresado por el usuario. El intervalo del TIP es 8 ó más periodos del clock, ingresados por el usuario.

En el CIP se identifican el registro de enmascaramiento de interrupciones (CIP.Mascara) y un registro de control. La Máscara del CIP es un registro de 8 bits. Este registro habilita los niveles de interrupción mayores que el del sP en ejecución.

La Máscara del procesador es un registro de 8 bits que mantiene los bits que se habilitan para los dispositivos autorizados a interrumpir al GASP.

La habilitación y deshabilitación general de interrupciones se realiza con un semáforo del .NET; el cual se deshabilita al entrar a las zonas críticas del GASP y se habilita al salir de ellas.

El bit 0 de todas las máscaras corresponde al nivel de máxima prioridad. El concepto de estas máscaras se abstrae y se define en la clase MascaraInt.

Un área de memoria para almacenar una variable entera a compartirse entre subprocesos.

Los dispositivos habilitados solicitan interrupción a través de instancias de la clase Interrupción que lo tramitan hacia el GASP.

5.1.2 CONTROL DE VELOCIDAD DE EJECUCION DE SPP.

El GASP se ejecuta dentro de un proceso sobre el .NET y el respectivo sistema operativo. Al crearse el proceso del GASP (PGA), también se crea su sP principal que es el único existente en el PGA hasta que se crean otros sPP. En este sP principal se inicializa el GASP, se crean y se arrancan los sPP de la configuración inicial del GASP y se crean y presentan las interfaces gráficas iniciales. Posteriormente, durante la vigencia del PGA, este mismo sP principal responde a las acciones del usuario sobre las interfaces y actualiza dichas interfaces (crea, actualiza y destruye formularios y sus elementos). Los otros sPP ejecutan tareas de cómputo, de atención de interrupciones y de otro tipo.

Dentro del PGA, en cualquier instante de tiempo dado solo hay un sP ejecutándose a la vez (incluyendo el sP principal) o ninguno. Cuando ningún sP del PGA se ejecuta, un sP de otro programa se está ejecutando en el computador. El GASP no controla la velocidad de ejecución de su sP principal, éste siempre se ejecuta a la velocidad permitida por el computador. La velocidad de ejecución de los otros sPP es controlable por el usuario por medio del GASP.

El usuario establece el periodo del reloj de la base de tiempos del GASP entre 600 ó mas milisegundos. Se emplea este reloj y el evento autorestablecible (AutoResetEvent) del .NET para controlar la velocidad de ejecución de los sPP que gestiona el GASP. Cada sP gestionado por el GASP tiene un AutoResetEvent, el cual es establecido por el reloj y usado (restablecido) a lo largo del cuerpo del sP, definiendo de esta manera el avance en su ejecución. El reloj es suministrado solamente al sP que se encuentra en el estado de ejecución.

5.1.3 ACREDITACION DE EJECUCION REFERENCIAL DE SPP.

Como se especificó en el párrafo anterior, cada sP gestionado por el GASP tiene su objeto AutoResetEvent. Este objeto se usa (se referencia) en cada punto del sP donde se desea acreditar el avance de ejecución. Las referencias a este objeto pueden aislar, para fines de ejecución, instrucciones individuales o grupos de instrucciones en lenguaje C#.

Junto a cada referencia al objeto AutoResetEvent se inserta la acreditación visual, en términos de código y/o datos del programa fuente, del avance de ejecución del sP. De esta manera, se expresa la ejecución de lenguaje máquina en términos del lenguaje C#.

5.1.4 CONMUTACION DE SPP.

Dentro del PGA, como se describe en los dos párrafos anteriores, los sPP gestionados por el GASP se ejecutan uno a la vez, a una velocidad definible por el usuario. Los mismos sPP gestionados acreditan su avance de ejecución en términos de programa fuente. De esta manera, el proceso hardware del computador presentado en el párrafo 2.1.4, se emula por el PGA en términos del reloj del GASP que pulsa el avance de los sPP, uno a la vez, cuando se encuentran en el estado de ejecución.

La conmutación de sPP gestionados se realiza en una zona crítica del GASP protegida por un semáforo. Las interrupciones también respetan esta zona crítica del GASP. En la perspectiva del GASP, la conmutación de sPP gestionados siempre se empieza y se termina con variables de memoria consistentes. En consecuencia, la conmutación de sPP gestionados dentro de un GASP no requiere almacenamiento de contexto de sP, ni de recuperación.

Durante su vida, un sP gestionado puede pasar esencialmente por los mismos estados graficados en la figura 4.1, presentada en el capítulo anterior.

5.2 ARQUITECTURA.

El GASP esta compuesto de paquetes de clases (subsistemas, servicios o módulos), organizados en (dos) capas, dentro de su contexto.

5.2.1 CONTEXTO.

El contexto del GASP está definido por los sPP que gestiona, el usuario interesado y la plataforma .NET sobre la que se despliega, tal como se presenta en la figura 5.1.

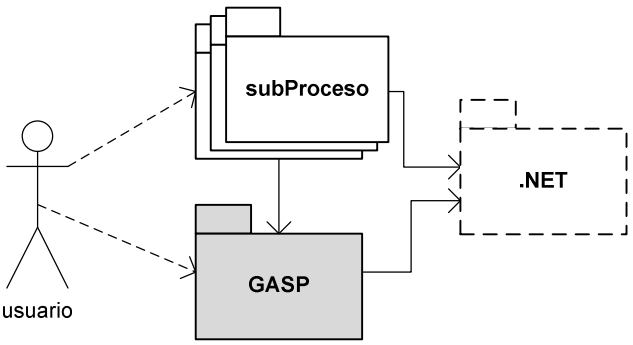


Figura 5.1 Contexto del gestor académico de subprocesos

5.2.2 CAPAS ESTRUCTURALES.

En la figura 5.2 se presenta la perspectiva estructural en dos capas del GASP y de los subprocesos que gestiona: capa de aplicación (servicios de aplicación) y capa de presentación (servicios de interfaz de usuario). Para destacar el GASP del resto del sistema, sus capas se han expresado en gris.

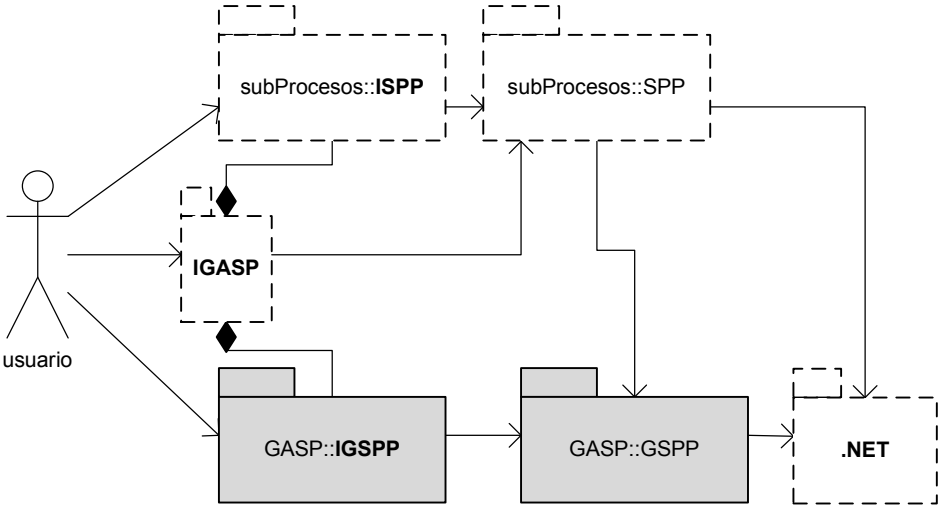


Figura 5.2 Arquitectura por capas del GASP en su contexto

El gestor de sPP (GSPP) constituye la capa de aplicación del GASP, está compuesto por las clases (atributos y operaciones) y sus relaciones a cargo de su funcionalidad dentro de sus requisitos no funcionales y sus restricciones. La interfaz gráfica de usuario del GSPP (IGSPP) constituye la capa de presentación del GASP, está compuesta por las clases y relaciones que gestionan la interfaz grafica de intermediación entre el usuario final y el GASP. La interfaz grafica de integración (IGASP) está compuesta por las clases que ponen a disposición del usuario final el GASP y los subprocesos gestionados. Análogamente, como se muestra en la figura, la gestión de subprocesos se implementa en dos capas, las cuales se tratan en el siguiente capítulo

5.2.3 PAQUETES DE CLASES.

Las clases dentro de las capas del GASP se organizan en paquetes, tal como que se muestran en al figura 5.3.

La interfaz del GASP es sencilla y atiende a un usuario a la vez, como consecuencia su capa de presentación se materializa en un paquete de clases, denominado IGestor. La capa de aplicación está constituida por cinco paquetes de clases, que se presentan a continuación.

La máscara de interrupciones define las características del registro de máscara de interrupciones empleado en el GASP y determinadas funciones relacionadas con prioridades software/hardware.

En el módulo de elementos básicos (EEBB) se encuentra la definición y la administración básica de los descriptores de subprocesos, de los buzones y de los

mensajes. En este módulo también se encuentra la mayor parte de la administración, por buzón, de las listas de mensajes esperando ser entregados y de los sPP esperando mensajes.

El módulo de listas se encarga, fundamentalmente, de manejar las listas de buzones y las listas de subprocesos: descriptores de sPP activos, sPP listos y sPP dormidos.

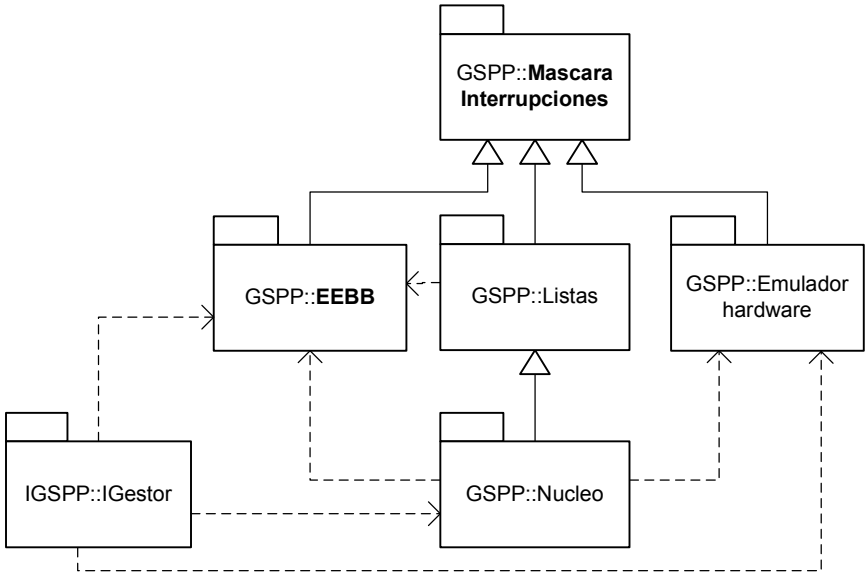


Figura 5.3 Arquitectura en paquetes del gestor académico de subprocesos

El Emulador de hardware (Hard) define las características de los dispositivos del computador importantes para el GASP: el CIP, el TIP, el reloj del GASP y la petición de interrupciones. También se emulan áreas de memoria, en los siguientes términos: un posición de memoria (memoria compartida), un búfer de una posición, un búfer de un número limitado de posiciones y un búfer de un número ilimitado de posiciones.

El núcleo, con la ayuda de los otros módulos, realiza operaciones de soporte y operaciones proporcionadas a sus usuarios.

Los principales usuarios del núcleo son los sPP que gestiona. Determinadas operaciones del núcleo, de elementos básicos y del emulador del hardware son accesibles por el usuario final por medio de interfaces gráficas. Estas interfaces gráficas son gestionadas por las clases del paquete de la capa de presentación del GASP.

5.3 CLASES Y OBJETOS DEL GASP.

El diseño de clases y objetos del GASP se presenta organizado por paquetes. Adicionalmente se presenta tareas específicas requeridas por el GASP y realizadas por subprocesos dedicados. Los diagramas de clasificadores (básicamente clases) constituyen el principal medio de expresión del diseño, se recurren a los diagramas de comportamiento de objetos solamente cuando lo ameritan. Los diagramas se presentan en UML empleando el modelador Microsoft Visio 2007 y considerando el lenguaje de programación C#. En estos diagramas estructurales, cada clasificador acredita sus características principales: atributos, operaciones y relaciones. Para mejor comprensión, los diagramas pueden omitir clasificadores (diferentes de clases) de su paquete e incluir clasificadores de otros paquetes. En el texto del párrafo se describen las características más importantes de cada clasificador del paquete y para determinadas operaciones se presentan su firma (especificación, cabecera) completa.

5.3.1 MASCARA DE INTERRUPCIONES.

Este paquete contiene una clase y un tipo de dato, tal como se muestra en el diagrama de la figura 5.4.

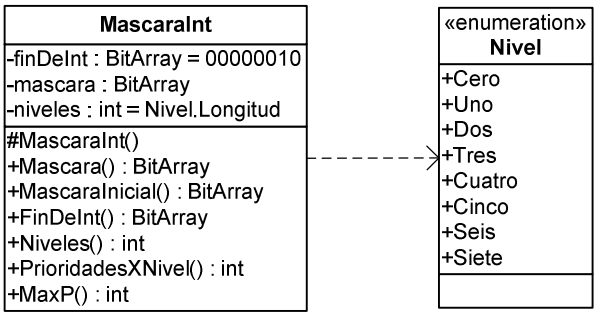


Figura 5.4 Clasificadores de Máscara de interrupciones

Nivel.

Tipo enumerado que identifica los niveles de interrupción hardware disponibles, el de mayor prioridad es el nivel Cero.

MascaraInt.

Clase con tres atributos, un constructor y seis propiedades:

- **mascara**, concreta el concepto de máscara de interrupción como un arreglo de bits (el bit del lado derecho, bit Cero, tiene la mayor prioridad).

- **finDeInt**, representa el reconocimiento de la interrupción del CIP (fin de interrupción del CIP).
- **niveles**, almacena el número de niveles de interrupción disponibles y define longitud de mascara y finDeInt.
- El **constructor** configura mascara y finDeInt.
- **Mascara**, propiedad que facilita y establece mascara.
- **MascaraInicial**, propiedad que facilita una configuración de mascara.
- **FinDeInt**, propiedad que facilita finDeInt.
- **PrioridadesXnivel**, propiedad que facilita el número de prioridades software correspondiente a un nivel de interrupción.
- **MaxP**, propiedad que facilita el valor numérico de la máxima prioridad software.

5.3.2 EMULADOR HARDWARE.

Para emular los dispositivos del computador que forma parte del GASP, este paquete contiene: una interfaz, siete clases, un tipo de datos y seis delegados. La interfaz, las clases y el tipo de datos se muestran en el diagrama de la figura 5.5.

dirES.

Tipo enumerado que define las direcciones para controlar el CIP: una para reconocer interrupciones y la otra para establecer la máscara de interrupciones.

IValor.

Declara una propiedad de lectura y escritura, implementada por las clases que emulan memoria.

CIP.

Clase que hereda la definición y comportamiento de su registro de máscara de interrupciones. Además, incorpora un atributo y dos operaciones:

- **unRegControl**, representa un registro de control del CIP direccionado cuando se reconocen interrupciones.
- El constructor, inicializa unRegControl.
- **this**, operación con índice que manipula los registros mascara y unRegControl del CIP. La firma de esta operación es la siguiente: `internal BitArray this[dirES dir] { get;set;}.`

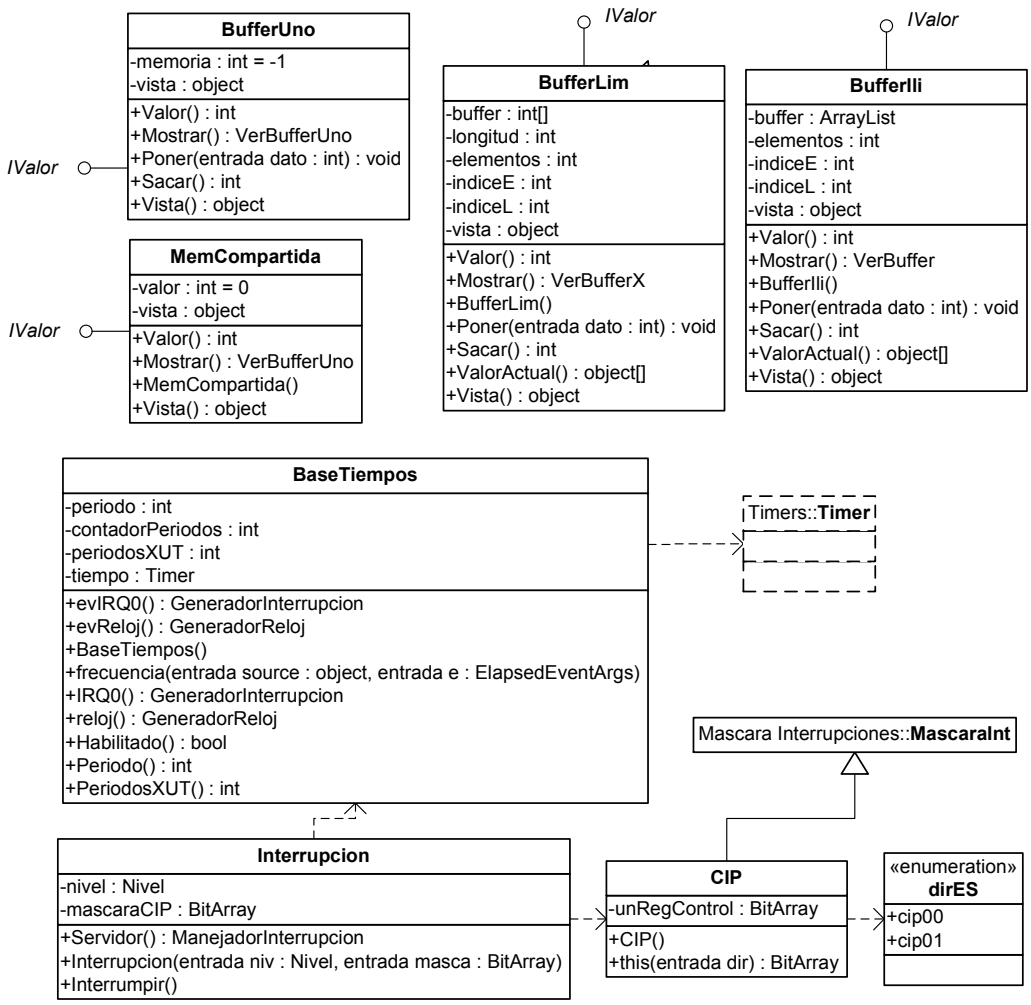


Figura 5.5 Dispositivos del computador emulados por el GASP

BaseTiempos.

Clase con cuatro atributos, dos eventos, un constructor, un método - procesador de eventos y cinco propiedades:

- **periodo**, número de milisegundos que el objeto Timer demora en generar cada evento para el objeto BaseTiempos.
- **periodosXUT**, múltiplo de periodo que define el intervalo del TIP que constituye la unidad de tiempo del GASP.
- **contadorPeriodos**, variable de trabajo para implementar el intervalo del TIP.
- **evReloj**, evento que produce el objeto BaseTiempos para definir el reloj del GASP.
- **evIRQ0**, evento que produce el objeto BaseTiempos, que representa la interrupción hardware del TIP y que define las unidades de tiempo del GASP.
- El constructor inicializa los atributos y configura el objeto Timer.
- **frecuencia**, procesador de eventos del objeto Timer que produce los eventos evIRQ0 y evReloj, emulando tanto el reloj del computador con evReloj (reloj del

GASP), como las interrupciones del TIP con evIRQ0 (unidades de tiempo del GASP). La firma del procesador es la siguiente: `private void frecuencia(objectsource,System.Timers.ElapsedEventArgs e).`

- **IRQ0**, propiedad para vincularse al evento evIRQ0.
- **reloj**, propiedad para vincularse al evento evReloj.
- **Habilitado**, propiedad para habilitar y deshabilitar el objeto Timer,
- **Periodo**, propiedad para facilitar y establecer periodo.
- **PeriodosXUT**, propiedad para facilitar y establecer periodosXUP.

Interrupción.

Clase que define la interrupción como un objeto, solicitada por un periférico y tramitada al Núcleo para su proceso. La clase tiene tres atributos, un constructor y un método – procesador de eventos:

- **nivel**, mantiene el nivel de interrupción del dispositivo.
- **mascaraCIP**, referencia el registro de mascara de interrupción del CIP.
- **Servidor**, evento que produce el objeto Interrupción para solicitar interrupción al Núcleo.
- El constructor recibe el nivel de interrupción del dispositivo y la referencia de la máscara del CIP y los almacena en los respectivos atributos.
- **Interrumpir**, procesador del evento evIRQ0 del TIP de la BaseTiempos.

IValor.

Interfaz que define la propiedad **Valor** con acceso de lectura y escritura. Esta interfaz es implementada por las siguientes cuatro clases que emulan áreas de memoria.

MemoriaCompartida.

Clase que define una palabra de memoria, compartida entre subprocesos gestionados. Esta clase tiene dos atributos, un constructor, un evento y dos propiedades.

- **valor**, representa la palabra de memoria.
- **vista**, referencia a formulario (interfaz gráfica) para visualizar palabra.
- **Mostrar**, evento que publica el valor de la palabra para visualización.
- El constructor inicializa la palabra de memoria.
- **Vista**, propiedad que facilita y establece la referencia a interfaz gráfica para la palabra de memoria.

- **Valor**, propiedad que devuelve y establece el valor de la palabra de memoria. Esta propiedad también lanza el evento Mostrar.

BufferUno.

Clase que emula un búfer que tiene una memoria (palabra). La clase tiene dos atributos, un evento, dos propiedades y dos métodos:

- **memoria**, representa el almacenamiento del búfer.
- **vista**, referencia a formulario (interfaz gráfica) para visualizar memoria.
- **Mostrar**, evento que publica el valor de memoria para visualización.
- **Vista**, propiedad que facilita y establece la referencia a interfaz gráfica para la memoria.
- **Valor**, propiedad que devuelve y establece el valor de la memoria. Esta propiedad también lanza el evento Mostrar.
- **Poner**, método que establece valor de la memoria.
- **Sacar**, método que obtiene el valor de la memoria.

BufferLim.

Clase que emula un búfer de un número limitado de memorias (palabras). La clase tiene seis atributos, un constructor, un evento, tres propiedades y dos métodos:

- **buffer**, arreglo de enteros que representa el almacenamiento del búfer.
- **longitud**, dimensión del búfer.
- **elementos**, cantidad de elementos disponibles en el búfer.
- **índiceE**, apunta la siguiente palabra a escribirse del búfer.
- **indiceL**, apunta la siguiente palabra a leerse del búfer
- **vista**, referencia a formulario (interfaz gráfica) para visualizar el búfer.
- **Mostrar**, evento que publica una palabra del búfer para visualización.
- El constructor inicializa los atributos.
- **Vista**, propiedad que facilita y establece la referencia a interfaz gráfica del búfer.
- **Valor**, propiedad que devuelve y establece la longitud del búfer. Esta propiedad lanza el evento Mostrar cuando se establece una nueva longitud.
- **ValorActual**, propiedad que devuelve todo el contenido del búfer.
- **Poner**, método que escribe un dato en el búfer y lo publica por medio del evento Mostrar.
- **Sacar**, método que obtiene el dato del búfer y lo publica por medio del evento Mostrar.

BufferIli.

BuferIli es análoga a la clase BufferLim, a diferencia de que no requiere el atributo longitud.

GeneradorReloj.

Delegado que define el tipo del evento evReloj. Su firma es la siguiente: `delegate void GeneradorReloj().`

GeneradorInterrupcion.

Delegado que define el tipo del evento evIRQ0. Su firma es la siguiente: `delegate void GeneradorInterrupcion().`

ManejadorInterrupcion.

Delegado que define el tipo del evento Servidor. La firma de este delegado es la siguiente: `delegate void ManejadorInterrupcion(Nivel nivel).`

VerBuffer.

Delegado que define el tipo del evento Mostrar de la Clase BufferIli. Su firma es la siguiente: `public delegate void VerBuffer(string mov, string nomSubP, int dato, int indice, int elementos).`

VerBufferUno.

Delegado que define el tipo del evento Mostrar de la Clase BufferUno. Su firma es la siguiente: `public delegate void VerBufferUno(string mov, string nomSubP, int mem).`

VerBufferX.

Delegado que define el tipo del evento Mostrar de la Clase BufferLim. Su firma es la siguiente: `public delegate void VerBufferX(string mov, string nomSubP, int dato, int indice, int elementos).`

5.3.3 ELEMENTOS BASICOS.

El paquete agrupa tres clases, tres tipos de datos y tres delegados. Los elementos básicos se refieren al descriptor de subproceso (BCP), al buzón y al mensaje, los cuales se definen en sus respectivas clases. Los otros clasificadores soportan las definiciones de los elementos básicos. En el diagrama de la figura 5.6 se presentan las clases y los tipos de datos de este paquete.

PrcocesoEstado.

Tipo enumerado que identifica los estados de un subproceso.

ProcesoTipo.

Enumerado que identifica los tipos de subprocesos.

MensajeTipo.

Enumerado que identifica los tipos de mensajes soportados.

Mensaje.

Clase que describe el mensaje con tres atributos, cuatro constructores y tres propiedades:

- **siguiente**, referencia para organizar lista de mensajes o para indicar si el mensaje ha sido consumido o esté pendiente de proceso.
- **tipo**, valor enumerado que indica el tipo de mensaje.
- **cuerpo**, referencia a objeto que contiene la información del mensaje.
- Los constructores permiten instanciar mensajes a partir de diferentes configuraciones de parámetros suministrados. Las firmas son las siguientes: `Mensaje(Mensaje sig, MensajeTipo tip, object cuer)`, `public Mensaje():this(null, MensajeTipo.Normal, null)`, `public Mensaje(object cuerpoRef):this(null, MensajeTipo.Normal, cuerpoRef)` y `public Mensaje(Mensaje mje)`.
- **Siguiente**, propiedad que facilita y establece el siguiente mensaje.
- **Tipo**, propiedad que facilita y establece el tipo de mensaje.
- **Cuerpo**, propiedad que facilita y establece el cuerpo del mensaje.

Buzón.

Clase que describe el buzón con ocho atributos, un constructor y cinco propiedades. Además, esta clase se ha sobrecargado seis métodos para gestionar una cola de mensajes, cinco métodos para gestionar la fracción de lista de sPP que esperan mensaje y seis mensajes para publicar actualización de ambas listas.

- **generadorId**, mantiene el número de mensajes creados.
- **buzónId**, número que identifica el mensaje.
- **primerMensaje**, inicio de cola de mensajes.
- **ultimoMensaje**, fin de cola de mensajes.
- **primerProceso**, inicio de fracción de lista de sPP que esperan mensaje.

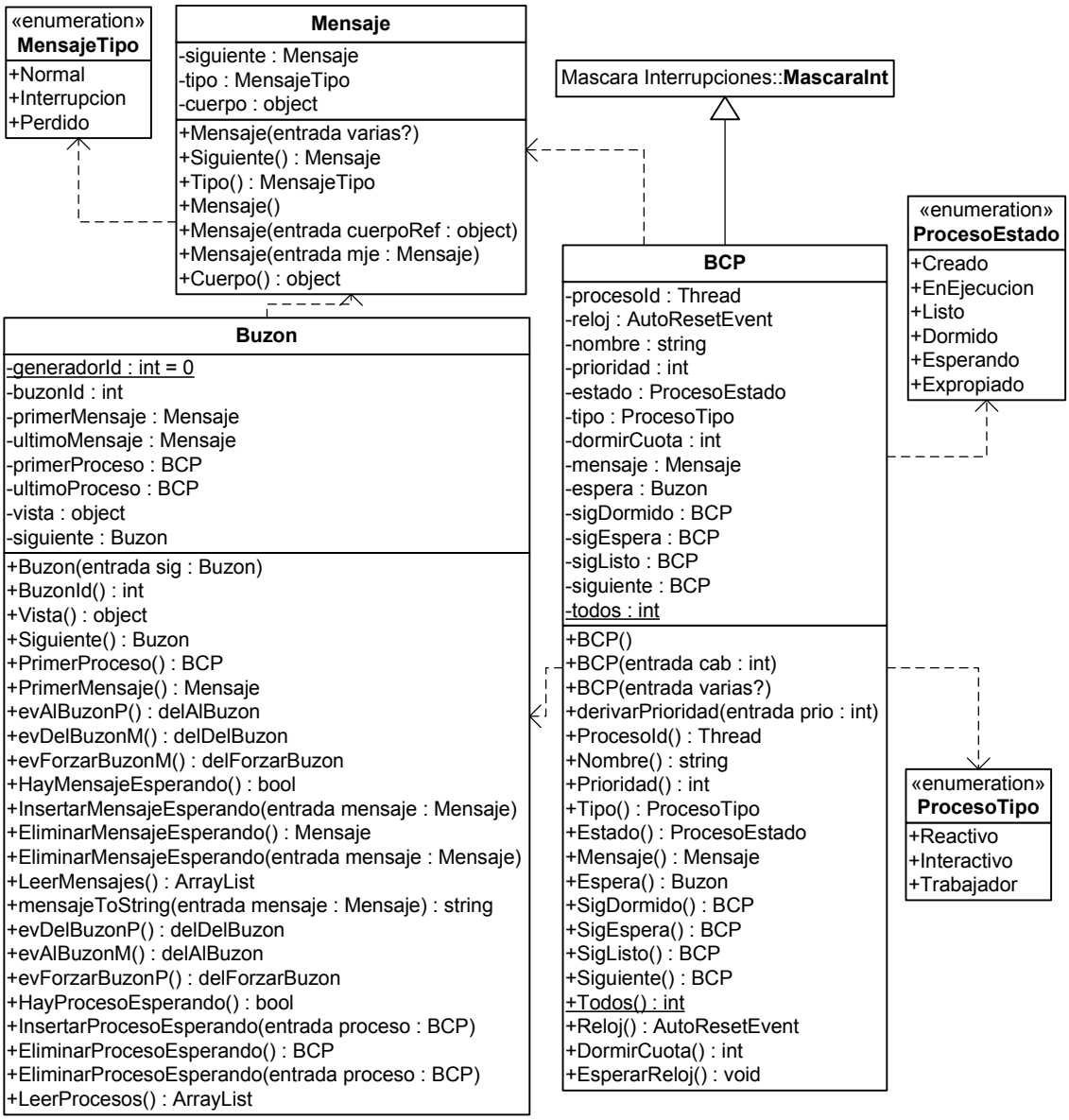


Figura 5.6 Clasificadores de elementos básicos del GASP

- **ultimoProceso**, fin de fracción de lista de sPP que esperan mensaje.
- **vista**, referencia a su interfaz visual.
- **siguiente**, referencia al siguiente buzón.
- El constructor configura el buzón creado.
- **evAlBuzonM** , evento para publicar mensaje insertado.
- **evDeBuzonM**, evento para publicar mensaje eliminado.
- **evForzarBuzonM**, evento para publicar mensaje eliminado, en otro orden.
- **evAlBuzonP**, evento para publicar subproceso insertado.
- **evDelBuzonP**, evento para publicar subproceso eliminado.
- **evForzarBuzonP**, evento para publicar subproceso eliminado, en otro orden.

- **BuzonId**, propiedad que facilita identificación de buzón.
- **Vista**, propiedad que facilita y establece su interfaz visual.
- **Siguiente**, propiedad que facilita y estables siguiente mensaje.
- **PrimerProceso**, propiedad que devuelve el inicio de la fracción de la lista de sPP que esperan mensaje en el buzón.
- **PrimerMensaje**, propiedad que devuelve el inicio de la cola de mensajes.
- **HayMensajesEsperando**, método que verifica existencia de cola de mensajes.
- **InsertarMensajeEsperando**, inserta mensaje dado en cola. Este método publica evento evAlBuzonM.
- **EliminarMensajeEsperando**, método que elimina mensaje de cola y publica evento evDelBuzonM. Su sobrecarga interpreta la cola como lista y elimina el mensaje dado, La sobrecarga publica el evento evForzarBuzonM.
- **LeerMensajes**, método que lee la lista de mensajes y lo devuelve en un arreglo.
- **MensajeToString**, método que convierte el mensaje dado a una cadena.
- **HayProcesoEsperando**, método que verifica existencia de cola de sPP en el buzón.
- **InsertarProcesoEsperando**, inserta sP dado en la cola de sPP del buzón. Este método publica evento evAlBuzonP.
- **EliminarProcesoEsperando**, método que elimina sP de cola de sPP del buzón y publica evento evDelBuzonP. Su sobrecarga interpreta la cola como lista y elimina el sP dado. La sobrecarga publica evento evForzarBuzonP.
- **LeerProcesos**, método que lee la cola de sPP del buzón y lo devuelve como un arreglo.

BCP

Esta clase hereda su máscara de MascaraInt y describe particularmente el subproceso con catorce atributos, tres constructores, catorce propiedades y dos métodos:

- **procesoId**, objeto que identifica el subproceso.
- **reloj**, objeto que identifica el reloj de ejecución del subproceso.
- **nombre**, cadena con nombre de subproceso
- **prioridad**, del subproceso: numero entre 0 y 255.
- **estado**, valor enumerado con estado de subproceso
- **tipo**, valor enumerado con tipo de subproceso
- **dormirCuota**, tiempo por dormir del sP, en unidades del GASP.
- **mensaje**, referencia de mensaje entregado a subproceso.

- **espera**, buzón donde el subproceso se encuentra esperando.
- **sigDormido**, siguiente subproceso dormido.
- **sigEspera**, siguiente subproceso en espera.
- **sigListo**, siguiente subproceso listo.
- **siguiente**, BCP en la lista de todos los subprocesos.
- **todos**, mantiene la cuenta de los sPP creados.
- Los constructores cumplen diferentes funciones: el primero inicializa todos, el segundo crea seudo subprocesos (cabecera de listas de sPP listos y sPP dormidos) y el tercero crea subprocesos. La firma del tercer constructor es la siguiente: `internal BCP(Thread proc,int prio,string nomb,BCP sig).`
- **ProcesoId**, propiedad que facilita el identificador del subproceso.
- **Reloj**, propiedad que facilita y asigna el objeto - reloj de ejecución del subproceso.
- **Nombre**, propiedad que facilita y establece el nombre del subproceso
- **Prioridad**, propiedad que facilita y establece la prioridad del subproceso entre 0 y 255.
- **Estado**, propiedad que facilita y establece el estado del subproceso
- **Tipo**, propiedad que facilita el tipo del subproceso
- **DormirCuota**, propiedad que facilita y establece el tiempo por dormir del subproceso, en unidades del GASP.
- **Mensaje**, propiedad que facilita y establece mensaje entregado a subproceso.
- **Espera**, propiedad que facilita y establece buzón donde el subproceso espera mensaje.
- **SigDormido**, propiedad que facilita y establece siguiente subproceso dormido.
- **SigEspera**, propiedad que facilita y establece siguiente subproceso en espera.
- **SigListo**, propiedad que facilita y establece siguiente subproceso listo.
- **Siguiente**, propiedad que facilita y estables siguiente BCP en la lista de todos los subprocesos.
- **Todos**, propiedad que facilita el número de subprocesos creados.
- **derivarPrioridad**, método que determina el tipo y la mascara de interrupción del subproceso a partir de su prioridad recibida.
- **EsperarReloj**, método que suspende la ejecución del subproceso hasta que recibe una señal del reloj del GASP.

delAlBuzon.

Delegado que define un tipo de evento para adicionar a la interfaz gráfica del buzón, un mensaje o un subproceso desde su respectiva cola en el buzón. Su firma es la siguiente: `public delegate void delAlBuzon(string s).`

delDelBuzon.

Delegado que define un tipo de evento para retirar de la interfaz gráfica del buzón, un mensaje o un subproceso. Su firma es la siguiente: `public delegate void delForzarBuzon(string s).`

delForzarBuzon.

Delegado que define un tipo de evento para retirar en forma especial de la interfaz gráfica del buzón, un mensaje o un subproceso. Su firma es la siguiente: `public delegate void delDelBuzon().`

5.3.4 LISTAS DE ELEMENTOS BASICOS.

Los clasificadores de este paquete manejan las listas de subprocesos, a excepción de los que esperan mensajes, y la lista de buzones; así como determinados aspectos de estas listas para comunicarlos a su respectiva su interfaz gráfica. El paquete contiene una clase, y cuatro delegados.

Listas.

Esta clase gestiona las listas: BCPs (todos los subprocesos), sPP listos y sPP dormidos. También, recopila información sobre estas listas a pedido de la respectiva interfaz gráfica. Esta clase dispone de la mascara de interrupción por herencia y define: quince atributos, dos constructores, cuatro eventos, catorce propiedades y veintiséis métodos:

- **enEjecucion**, referencia a subproceso en ejecución.
- **procesos**, lista de todos los sPP.
- **listos**, cabecera de lista de sPP listos.
- **dormidos**, cabecera de lista de sPP dormidos.
- **proNombre**, nombre de la lista de todos los sPP.
- **lisNombre**, nombre de la lista de sPP listos.
- **dorNombre**, nombre de la lista de sPP dormidos.
- **buzNombre**, nombre de la lista de buzones.
- **verElProceso**, indica solicitud para monitorear un sP.
- **elProcesoNombre**, nombre de sP a monitorear.

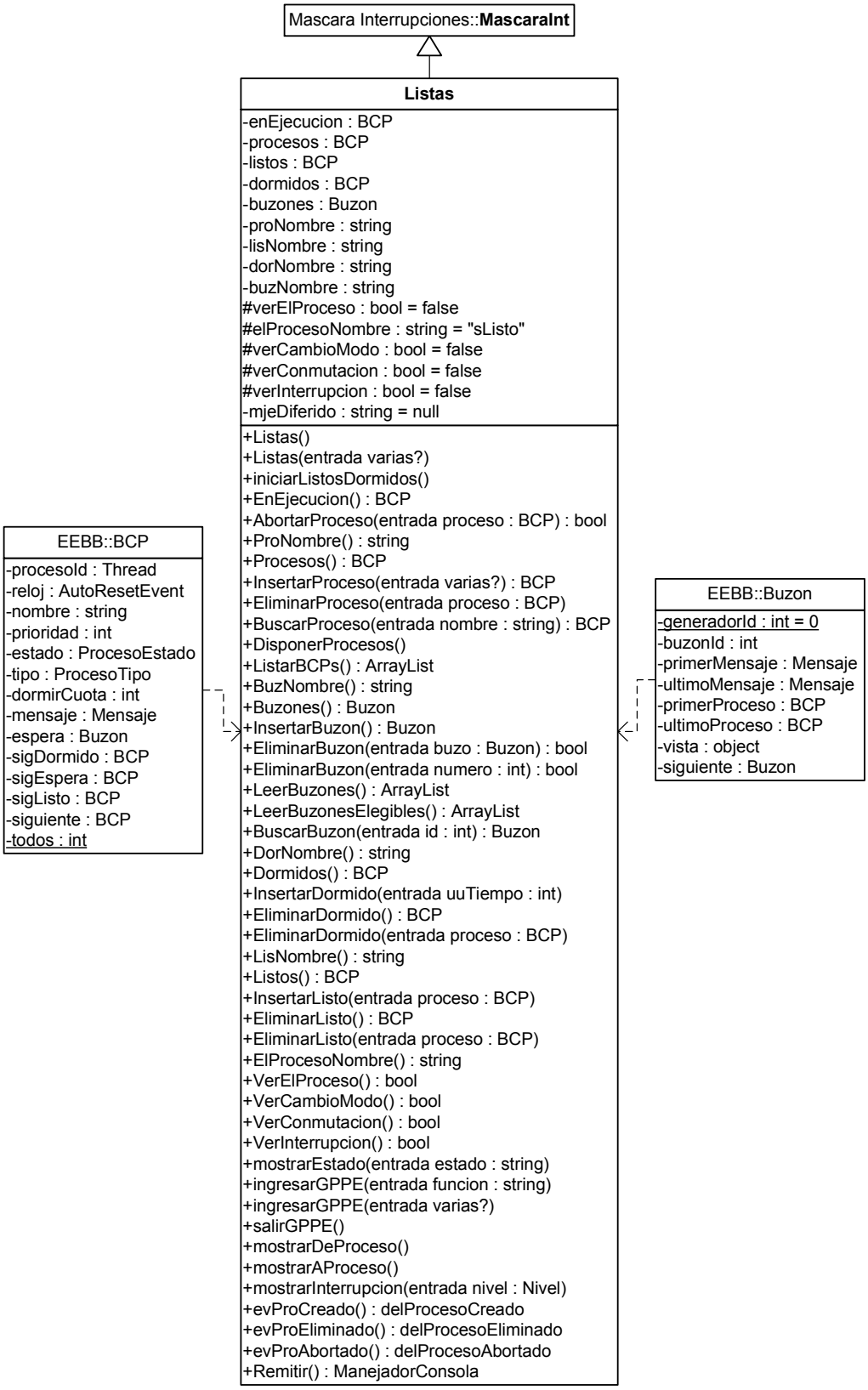


Figura 5.7 Clasificadores que manejan listas de elementos básicos

- **buzones**, lista de buzones.
- **verCambioModo**, indica solicitud para monitorear paso de sP a GASP y viceversa.
- **verConmutacion**, indica solicitud para monitorear conmutación de sPP.
- **verInterrupcion**, indica solicitud para monitorear interrupciones.
- **mjeDiferido**, cadena de trabajo para armar textos para consola de sPP.
- Los constructores realizan la configuración inicial de listas con nombres por defecto o recibiendo todos los nombres de lista. Sus firmas son las siguientes: `internal Listas():this("procesos","listos","dormidos", "buzones")` y `internal Listas(string proc, string list, string dorm, string buzo).`
- **evProCreado**, evento para publicar que sP se ha creado.
- **evProEliminado**, evento para publicar que sP se ha eliminado.
- **evProAbortado**, evento para publica que sP se ha abortado.
- **Remitir**, evento para publicar información de sPP a consola.
- **ProNombre**, propiedad que facilita nombre de lista de todos los sPP.
- **LisNombre**, propiedad que facilita nombre de lista de sPP listos.
- **DorNombre**, propiedad que facilita nombre de lista de sPP dormidos.
- **BuzNombre**, propiedad que facilita nombre de lista de buzones.
- **EnEjecución**, propiedad que facilita y establece referencia al único sP en el estado de ejecución.
- **Procesos**, propiedad que facilita lista de todos los sPP.
- **Listos**, propiedad que facilita lista de sPP listos.
- **Dormidos**, propiedad que facilita de lista de sPP dormidos.
- **Buzones**, propiedad que facilita lista de buzones.
- **ElProcesoNombre**, propiedad que facilita y establece el nombre del sP a monitorear en la consola de sPP.
- **VerElProceso**, propiedad que facilita y establece la autorización para monitorear el sP establecida en la propiedad anterior.
- **VerCambioModo**, propiedad que facilita y establece la autorización para monitorear el paso de usuario a GASP y viceversa.
- **VerConmutacion**, propiedad que facilita y establece la autorización para monitorear la conmutación entre sPP.

- **VerInterrupcion**, propiedad que facilita y establece la autorización para monitorear las interrupciones.
- **iniciarListosDormidos**, método interno que crea cabecera de lista de sPP listos y cabecera de lista de sPP dormidos.
- **InsertarProceso**, método que inserta un nuevo descriptor de subproceso (BCP) en la lista de BCPs, por el inicio. Su firma es la siguiente: `internal BCP InsertarProceso( Thread procId, int prio, string nomb).`
- **EliminarProceso**, método que elimina el BCP dado de la lista de todos los sPP.
- **AbortarProceso**, método que aborta un sP en el entorno .NET y los elimina de las listas en las que se **encuentra**.
- **BuscarProceso**, método que busca el sP, por su nombre, en la lista de todos los sPP y devuelve su BCP.
- **DisponerProcesos**, método que aborta en el entorno .NET todos los subprocesos sin modificar las listas.
- **ListarBCPs**, método que lee todos los sPP y los devuelve en un arreglo de cadenas.
- **InsertarListo**, método que inserta un sP dado en lista de sPP listos, antes del sPP de menor prioridad. El inicio de esta lista siempre está ocupado por el sP listo (incluyendo recién creados y expropiados) de mayor prioridad.
- **EliminarListo**, método que retira un sP de la lista de sPP listos. Este método tiene una sobrecarga. Sus firmas son las siguientes: `internal BCP EliminarListo(), internal void EliminarListo(BCP proceso).`
- **InsertarDormido**, método que inserta un sP en la lista de sPP dormidos, delante del sP que tiene mayor tiempo por dormir. Los sPP en esta lista están ordenados por el tiempo a dormir que les resta, el último sP es el que tiene el mayor tiempo por dormir. El algoritmo implementa tiempos relativos en lugar de tiempos absolutos. El tiempo recibido por el método está expresado en unidades de tiempo del GASP.
- **EliminarDormido**, método que retira un sP del inicio de la lista de sPP dormidos. Este método tiene una sobrecarga que retira un sP de cualquier lugar de la lista. Sus firmas son las siguientes: `internal BCP EliminarListo(), internal void EliminarListo(BCP proceso).`
- **InsertarBuzon**, método que crea e inserta un buzón por el inicio de la lista de buzones.

- **EliminarBuzon**, método que elimina un buzón dado de la lista de buzones. Este método tiene una sobrecarga que elimina un buzón de la lista, dado el identificador de buzón. Sus firmas son las siguientes: `internal bool EliminarBuzon(Buzon buzo)`, `internal bool EliminarBuzon(int numero)`.
- **LeerBuzones**, método que lee los identificadores de buzón de la lista de buzones y los devuelve como un arreglo de cadenas.
- **LeesBuzonesElegibles**, método que lee los identificadores de buzón de la lista de buzones, sin incluir los buzones dedicados a mensajes de interrupción.
- **BuscarBuzón**, método que busca un buzón, dado su identificador.
- **mostrarEstado**, método que publica evento Remitir con estado de sP.
- **ingresarGPPE**, método que publica evento Remitir, indicando el tránsito de sP en ejecución a una operación del GASP. Este método tiene una sobrecarga que recibe como parámetro el sP del cual se transita a GASP. Sus firmas son las siguientes: `protected void ingresarGPPE(string funcion)`, `protected void ingresarGPPE(string nombre, string funcion)`.
- **salirGPPE**, método que publica evento Remitir, indicando el tránsito del GASP a sP.
- **mostrarDeProceso**, método que publica evento Remitir, indicando inicio de conmutación de sPP.
- **mostrarAProceso**, método que publica evento Remitir, indicando la culminación de conmutación de sPP.
- **mostrarInterrupcion**, método que publica evento Remitir, indicando la interrupción sucedida.

delProcesoCreado.

Delegado que define un tipo de evento para publicar la creación de un sP. Su firma es la siguiente: `internal delegate void delProcesoCreado (string[] datos)`.

delProcesoEliminado.

Delegado que define un tipo de evento para publicar la eliminación de un sP. Su firma es la siguiente: `internal delegate void delProcesoEliminado (string nombre)`.

delProcesoAbortado.

Delegado que define un tipo de evento para publicar que un sP ha sido abortado. Su firma es la siguiente: `public delegate void delProcesoAbortado (string nombre).`

ManejadorConsola.

Delegado que define el tipo del evento Remitir para publicar dinámica de ejecución de sPP. Su firma es la siguiente: `public delegate void ManejadorConsola(string s).`

5.3.5 NUCLEO.

En núcleo contiene la clase GP (gestión de subproceso). GP realiza los servicios que brinda el GASP con la colaboración de los clasificadores de los otros paquetes y de los subprocesos del paquete Tareas Específicas. Las clases del GASP y sus relaciones se muestran en el diagrama de la figura 5.8. El diagrama presenta GP en detalle y los nombres de las clases de los otros paquetes.

Tal como se muestra en el diagrama, GP hereda la funcionalidad de Listas y particularmente define siete atributos, un constructor, tres propiedades y dieciocho métodos. La especificación de los métodos de GP se presenta dentro de las funciones generales del GASP, dadas en la especificación funcional del capítulo anterior. Los atributos y propiedades son los siguientes:

- **semGEsspp**, objeto semáforo .NET para protección de concurrencia del GASP y que implementa el modo de operación (usuario y GASP). Las operaciones del GASP pertinentes obtienen el semáforo (deshabilitan interrupciones) para ingresar a zonas críticas y lo liberan (habilitan interrupciones) al salir.
- **enGEsspp**, indica modo de operación: GASP (true) o usuario (false).
- **cip**, objeto CIP.
- **mensajeInterrupcion**, mensajes de interrupción.
- **buzonInterrupcion**, buzones de interrupción.
- **HardIRQ0**, objeto para captar interrupciones del nivel Cero.
- **bt**, objeto base de tiempos.
- **BT**, propiedad que facilita y establece la base de tiempos.
- **IRQ0**, propiedad que facilita el valor Nivel.Cero en forma numérica. Análogamente se puede proceder con los otros valores de este tipo enumerado.

- **BuzonIRQ0**, propiedad que facilita el buzón asociado con la interrupción nivel Cero. Análogamente se puede proceder con los buzones asociados a los otros niveles.

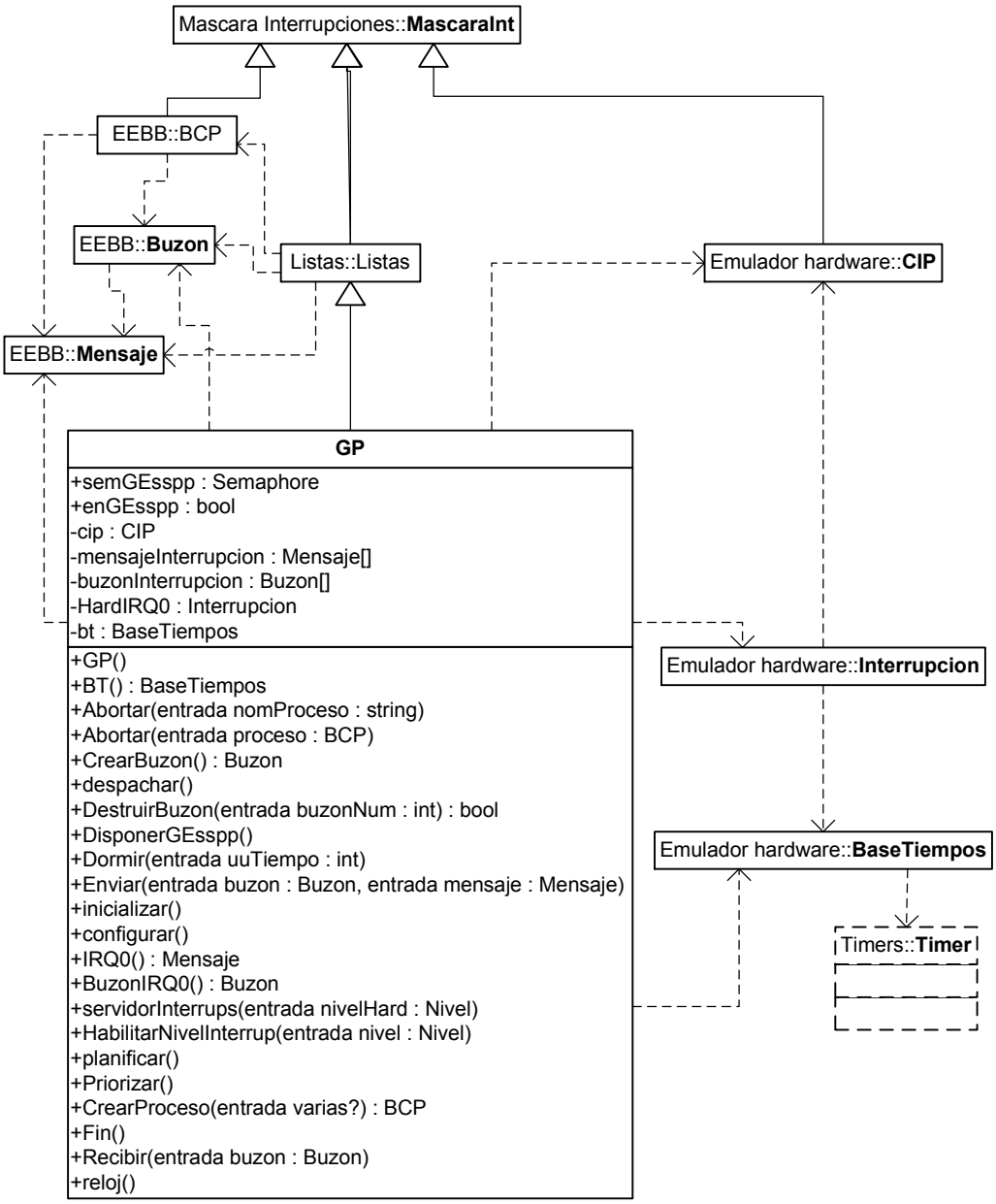


Figura 5.8 Diagrama de clases del GASP

5.3.5.1 Configuración inicial.

La operación interna CONFIGURACION INICIAL se realiza por medio de un constructor y dos métodos:

- En constructor, invoca a los métodos privados inicializar y configurar.

- **inicializar**, método privado que crea semáforo capturado, inicializa los atributos de la clase y libera semáforo.
- **configurar**, método privado que completa la configuración inicial del GASP, preparando los subprocesos sDespertador y sListo. El primero tiene la máxima prioridad en el GASP y atiende las interrupciones del TIP y el segundo tiene la mínima prioridad y siempre está listo para ejecutarse. El método termina planificando y despachando el subproceso de mayor prioridad.

5.3.5.2 Gestión de buzones.

Las operaciones de CREAR BUZON y DESTRUIR BUZON se realizan en esta clase por medio de los siguientes métodos:

- **CrearBuzon**, método que captura el semáforo, crea un buzón y libera el semáforo.
- **DestruirBuzon** método que captura el semáforo, elimina un buzón dado su identificador y el libera el semáforo.
- Además están accesibles los siguientes métodos para monitorear la lista de buzones: `public Buzon BuscarBuzon(int id)`, `public ArrayList LeerBuzones()` y `public ArrayList LeerBuzonesElegibles()`.

5.3.5.3 Planificación básica de subprocesos.

Las operaciones de PLANIFICAR SUBPROCESO y DESPACHAR SUBPROCESO se realizan por medio de los siguientes métodos:

- **planificar**, método privado que toma el primero de la lista de subprocesos listos y lo pasa al lugar del subproceso a ejecutarse, sin cambiar su estado. La planificación propiamente dicha se realiza al insertar un subproceso en la lista de subprocesos listos.
- **despachar**, método privado que toma el subproceso planificado para ejecución, prepara la máscara de interrupción de dicho subproceso y cambia su estado de listo a en ejecución (despacha). El método termina liberando el semáforo del GASP.
- **Priorizar**, método privado que captura el semáforo y luego invoca a planificar y despachar.

5.3.5.4 Interacción entre subprocesos.

Las operaciones de ENVIAR MENSAJE y RECIBIR MENSAJE se realizan por medio de los siguientes métodos:

- **Enviar**, método con dos parámetros: buzón y mensaje. Después de capturar semáforo, verifica si hay subproceso esperando en el buzón indicado. Si hay, entrega el mensaje al subproceso que espera y lo pasa a estado listo, sino, deposita el mensaje en el buzón. Finaliza invocando a planificar y despachar. Su firma es la siguiente: `public void Enviar(Buzon buzon, Mensaje mensaje).`
- **Recibir**, método con un parámetro: buzón. Después de capturar semáforo, verifica si hay mensaje esperando en el buzón indicado. Si hay, extrae el mensaje, sino, se inserta a la cola de espera de mensajes en el buzón. Finaliza invocando a planificar y despachar. Su firma es la siguiente: `public void Recibir(Buzon buzon).`

5.3.5.5 Gestión de interrupciones.

Las operaciones SERVIR INTERRUPCIONES y HABILITAR NIVEL DE INTERRUPCION se realizan por los siguientes métodos:

- **servidorInterrups**, método privado que recibe nivel de interrupción como parámetro y protege su cuerpo con un monitor de .NET. Expropia subproceso en ejecución, determina mensaje de interrupción y el buzón asociado sobre la base del nivel de interrupción, reconoce interrupción a dispositivo que la solicitó y verifica si mensaje anterior del mismo nivel. Si el mensaje anterior ha sido procesado, deposita el presente mensaje en el buzón con el método Enviar, sino marca el mensaje de interrupción como perdido y pasa a ejecución el subproceso listo de mayor prioridad con el método priorizar. Su firma es la siguiente: `void servidorInterrups(Nivel nivelHard).`
- **HabilitarNivelInterrup**, método que habilita el bit de la mascara de interrupciones del sistema, correspondiente al nivel dado como parámetro. Su firma es la siguiente: `internal void HabilitarNivelInterrup(Nivel nivel).`

5.3.5.6 Administración de subprocesos.

Las operaciones CREAR SUBPROCESO, FINALIZAR SUBPROCESO, ABORTAR SUBPROCESO, DORMIR SUBPROCESO y DESPERTAR SUBPROCESO se realizan por medio de las siguientes métodos:

- **CrearProceso**, método que recibe los parámetros del subproceso ya creado en .NET, captura el semáforo, crea su BCP en la lista de todos los subprocesos, inserta el BCP en la lista de subprocesos listos (aquí efectivamente se planifica) y libera el

semáforo. Su firma es la siguiente: `public BCP CrearProceso(Thread proc, int prio, string nomb)`

- **Fin**, método sin parámetros que realiza la operación FINALIZAR SUBPROCESO. Captura el semáforo, elimina el subproceso en ejecución (retira de la lista de listos y de todos los subprocesos), planifica y despacha.
- **Abortar**, método que aborta un subproceso identificado por el nombre del subproceso recibido como parámetro. Captura el semáforo y busca el subproceso. Si lo encuentra lo aborta, planifica y despacha. Sino, libera semáforo. La sobrecarga de esté método recibe como parámetro el BCP del subproceso. Sus firmas son las siguientes: `internal void Abortar(string nomProceso)` y `public void Abortar(BCP proceso).`
- **DisponerGEsspp**, método sin parámetros que elimina todos los subproceso del GASP. Captura el semáforo, deshabilita la base de tiempos, aborta todos los subprocesos a nivel de .NET (invocando al método interno DisponerProcesos) y libera el semáforo.
- **Dormir**, método que recibe un tiempo como parámetro y pone a dormir el subproceso en ejecución por ese tiempo. Captura el semáforo, retira el subproceso de la lista de listos y lo inserta en la lista dormidos (siempre y cuando las unidades de tiempo sea mayor que cero y menor 1001). Termina invocando a planificar y despachar.
- **Reloj**, método que procesa los eventos de la base de tiempos para definir el reloj del GASP. Su firma es la siguiente: `private void reloj().`
- La operación DESPERTAR SUBPROCESO se realiza por el subproceso spDespertar que se presenta en el paquete tareas específicas.

5.3.6 IGESTOR.

Este paquete contiene las clases e interfaces gráficas (formularios) que constituyen la capa de presentación del GASP. Por medio de estas interfaces, el interesado conduce la dinámica del GASP. Cuando se muestra información en los formularios que es producida como parte de los subprocesos gestionados, se emplea programación asíncrona. La explicación de las clases e interfaces se presenta en forma esencialmente gráfica. En una figura, la interfaz parcialmente superpone al diagrama UML que muestra la clase completamente especificada y relacionada a las clases pertinentes de la capa de aplicación. Estas interfaces son las siguientes:

Figura 5.9 Interfaz de usuario de la base de tiempos

Interfaz de la base de tiempos.

El formulario y la clase IUBaseTiempos que lo gestiona se presentan en la figura 5.9. La clase permite ver y establecer el periodo del reloj del GASP en milisegundos y el valor de las unidades de tiempo del GASP en periodos de este reloj, así como arrancar (reanudar) y parar (suspender) el funcionamiento del GASP y por, ende, del los subprocesos en gestión. El formulario de la figura 5.9 muestra que se está por arrancar el GASP con un reloj de 720 milisegundos y unidades de tiempos de 5760 milisegundos.

Interfaz de presentación de buzón.

La interfaz de buzón está compuesta por un formulario, servido por la clase IUBuzon, tal como se presenta en la figura 5.10. Esta interfaz solamente muestra los mensajes o los subprocesos depositados en el buzón visualizado. En la figura se muestra el buzón 8 con tres mensajes pendientes de ser entregados y el buzón 9 con dos subprocesos esperando mensaje. Esta interfaz suele asociarse a la interfaz gráfica de subprocesos gestionados.

En la presentación de los mensajes o subprocesos en el formulario, la clase IUBuzon emplea programación asíncrona.

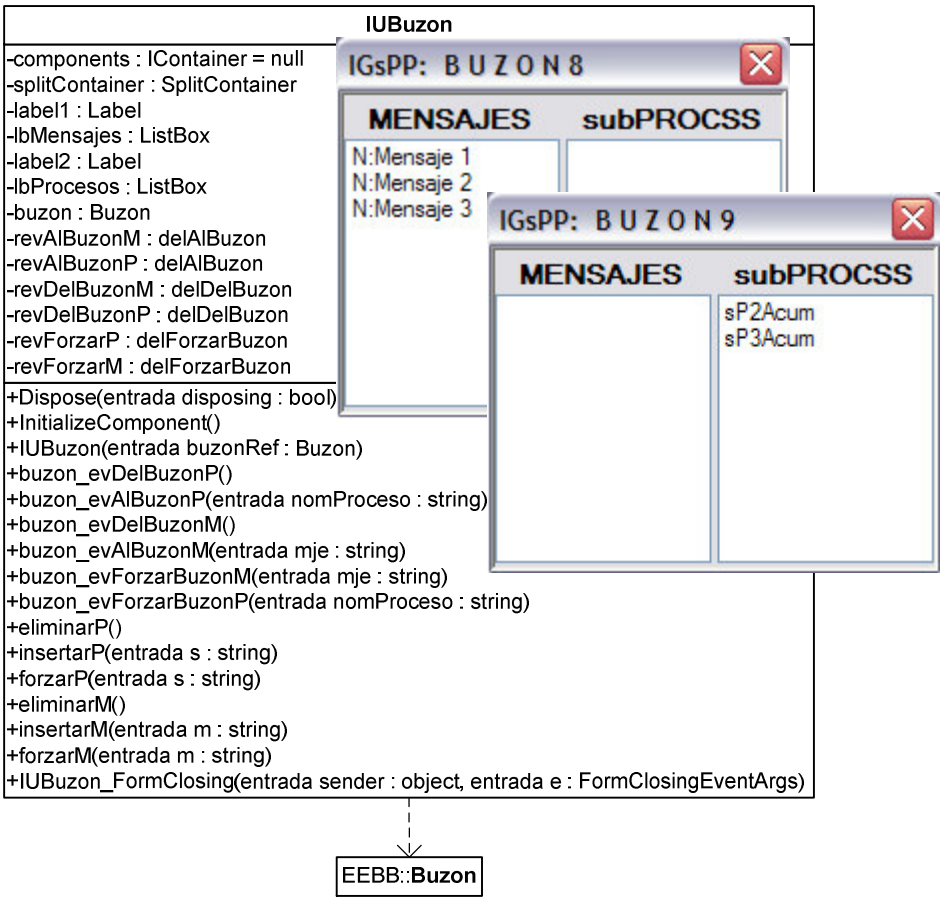


Figura 5.10 Interfaz de usuario de buzon

Interfaz de gestión de lista de buzones.

El usuario puede gestionar la lista de buzones por medio de esta interfaz con el servicio de la clase IUBuzones, como se aprecia en la figura 5.11. Inicialmente el formulario muestra la lista de buzones existentes, luego permite añadir nuevos buzones y eliminar los existentes. Los buzones asociados a los mensajes de interrupción no pueden ser eliminados.

En la figura 5.11, se aprecian tres buzones agregados: 8, 9 y10, así como el buzón 9 seleccionado.

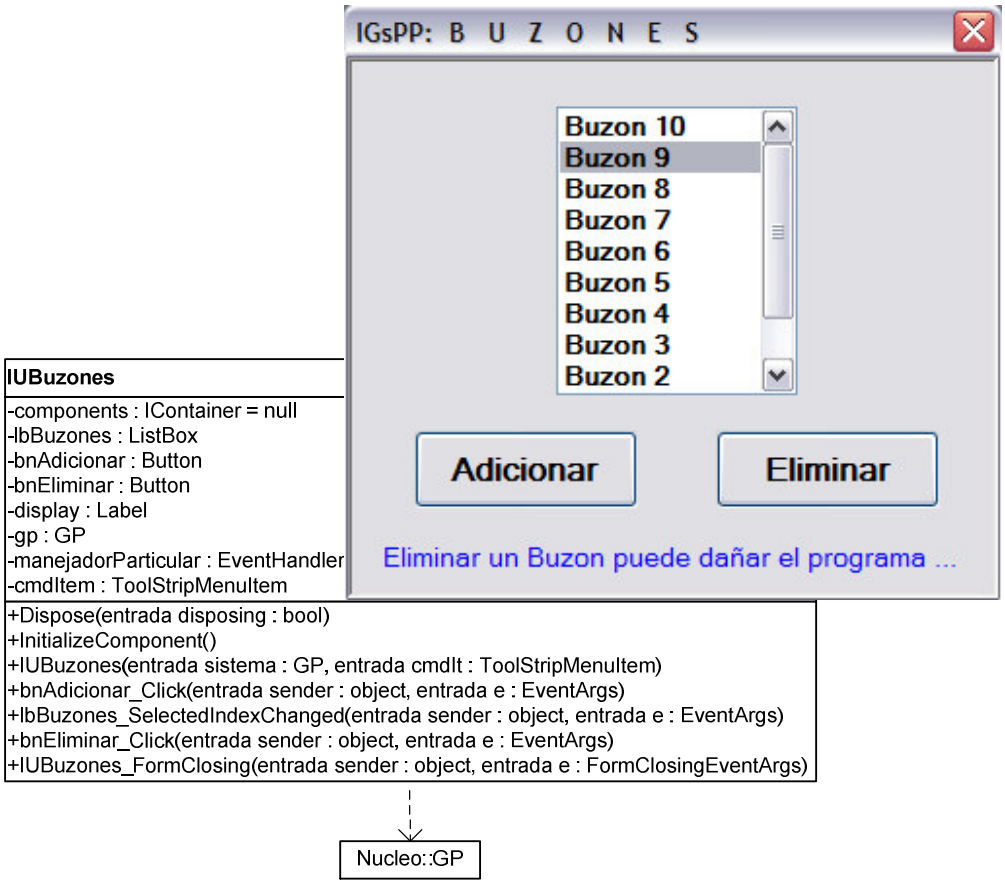


Figura 5.11 Interfaz de gestión de lista de buzones

Interfaz de gestión de mensajes.

Esta interfaz y su servicio IUMensajes se presentan en la figura 5.12.

Por medio de esta interfaz, el usuario accede a cada buzón de la lista de buzones y visualiza la cola de mensajes o la cola de subprocesos depositada en él; así mismo puede adicionar nuevos mensajes a este buzón y eliminar los mensajes existentes. Durante la adición nuevos de mensajes, el usuario elije el tipo de mensaje y, opcionalmente, ingresa información para el cuerpo del mensaje.

Las colas del buzón son procesadas por el GASP con el soporte de spMjeIn y spMjeOut, subprocesos gestionados ubicados en el paquete Tareas específicas. La clase IUMensajes recurre a la programación asíncrona para actualizar determinados elementos del formulario. En la figura 5.12 se aprecia el buzón 10 con una cola de cuatro mensajes esperando ser entregados.

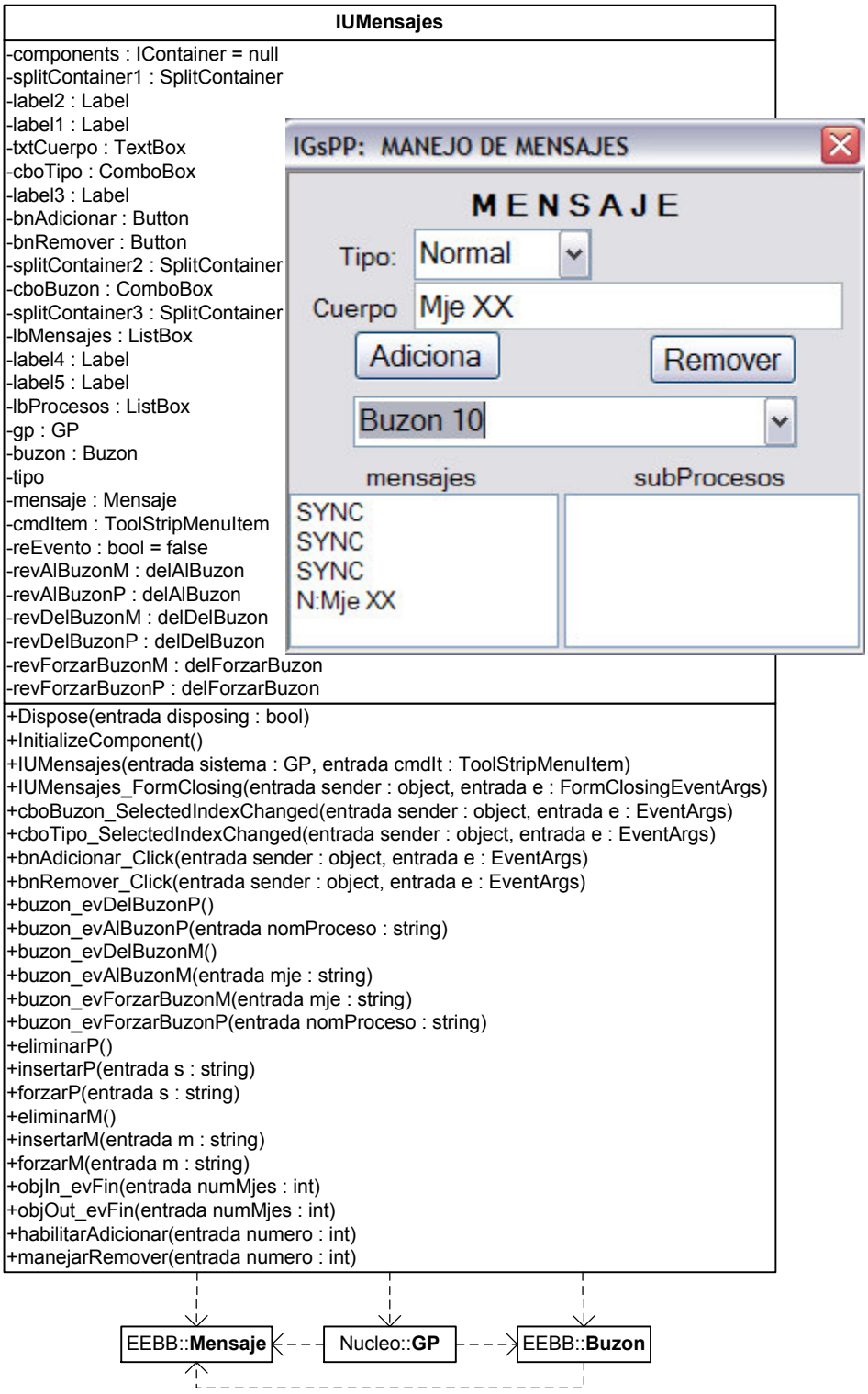


Figura 5.12 Interfaz de administración de mensajes por buzón

Consola del GASP.

En la consola del GASP, se puede observar: la evolución de un subproceso durante su vida en términos de sus estados, la interacción entre los subprocesos y el GASP (cambio de modo), la ocurrencia de interrupciones y la conmutación entre subprocesos.

La figura 513 muestra esta consola con su servicio IUConsola. El formulario cuenta con casillas de verificación para habilitar la característica deseada. La clase IUConsola recurre a la programación asincrónica.

En el formulario de la figura se aprecia información sobre cambios de modo de operación (usuario < == > GASP) y evolución de la vida del subproceso sListo, seleccionados por las respectivas casillas de verificación.

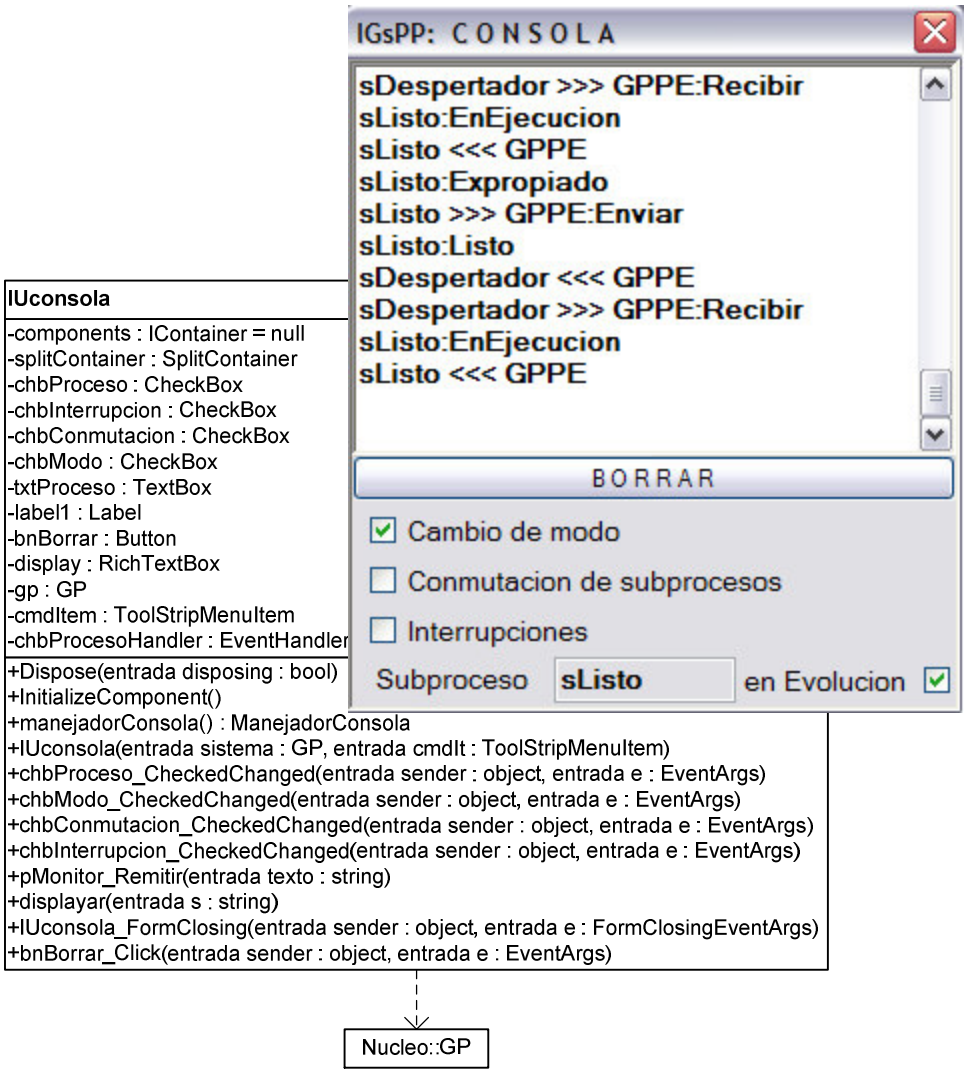


Figura 5.13 Consola de GASP

Monitor de lista de subprocesos.

Esta interfaz muestra la lista de subprocesos existentes. El formulario se actualiza dinámicamente a medida que se crean nuevos subprocesos o finalizan su ejecución. Además, la interfaz da la posibilidad al usuario de abortar un subproceso. El subproceso siempre listo y el subproceso despertador no son abortables. El Formulario y su servicio IUProcesos se presentan en la figura 5.14. La clase IUProceso recurre a la programación asincrónica para manejar el monitor. El formulario de la figura 5.14, se aprecian cinco subprocesos en el monitor: sListo, sDespertador, sP2Acum y sP3Acum.

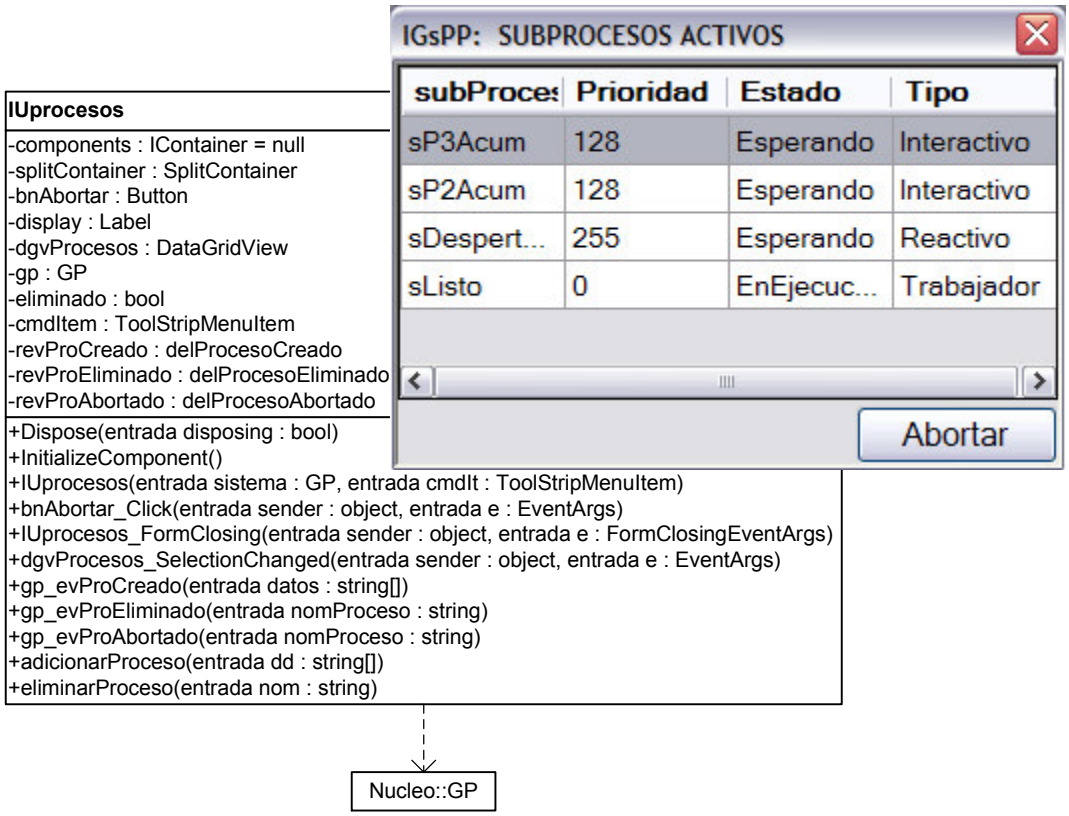


Figura 5.14 Monitor de subprocesos.

Interfaz de memoria compartida.

Esta interfaz visualiza la palabra de memoria compartida por subprocesos gestionados, mostrando el contenido de la palabra y los subprocesos que lo accedan. El formulario y la clase IUMemoria, que le da servicio, se muestran en la figura 5.14. Esta interfaz se asocia a la interfaces de los subprocesos que comparten la memoria.

La clase IUMemoria recurre a la programación asíncrona para gestionar su formulario. En el formulario de la figura 5.15, se aprecia que el subproceso sP2Acum ha leído de la palabra de memoria el valor 0 y ha escrito el valor 7 en ella.

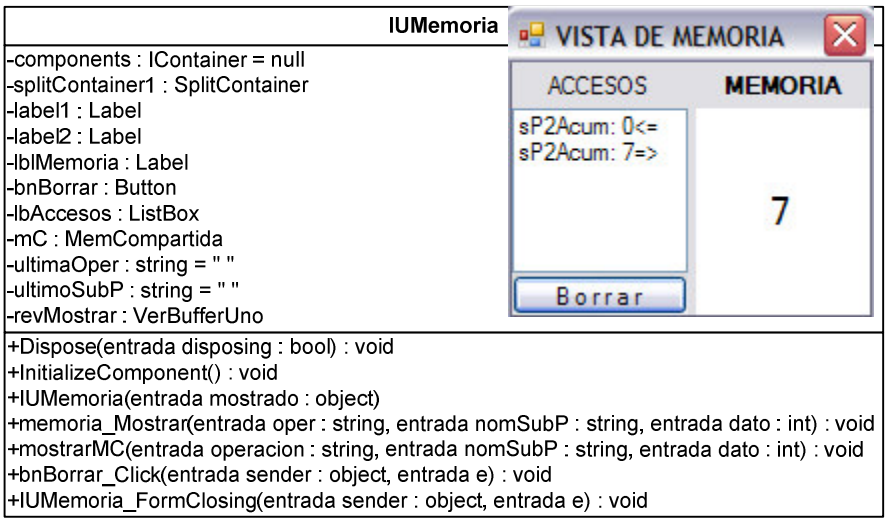


Figura 5.15 Interfaz de memoria compartida

Interfaz de búfer de una memoria.

El formulario que visualiza el búfer de una memoria y su servicio IUBufferUno se presentan en la figura 5.16. La clase emplea programación asíncrona. El formulario de la figura muestra escrituras del subproceso sP4Prod y lecturas del subproceso sP3Cons.

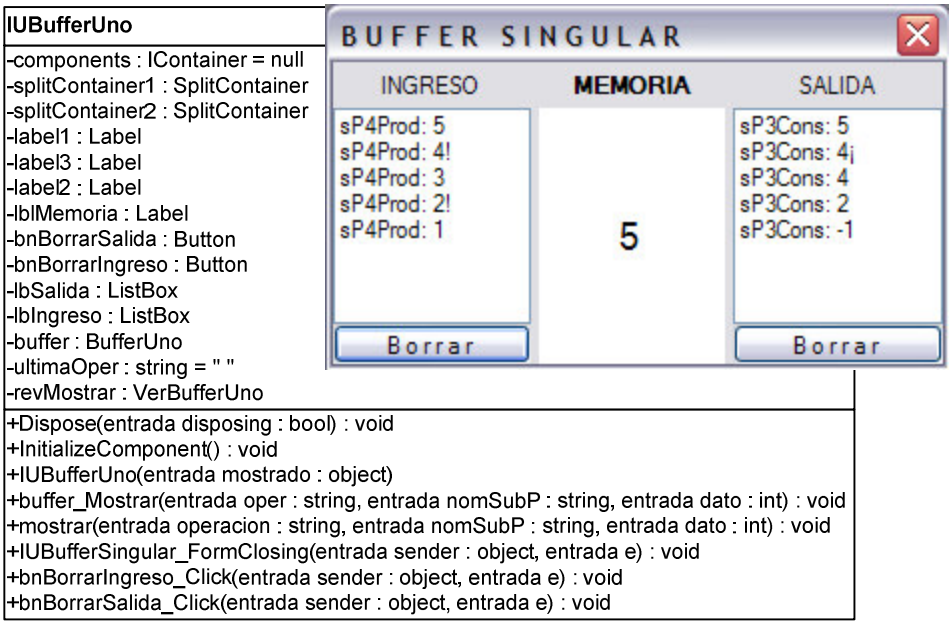


Figura 5.16 Interfaz de búfer de una memoria

Interfaz de búfer de número limitado de memorias.

El formulario que visualiza el búfer de memorias limitadas y su servicio IUBuffeX se presentan en la figura 5.17. La clase emplea programación asíncrona. El formulario de la figura muestra un búfer con tres memorias, en el cual el subproceso sP7Prod escribe y del cual el subproceso sP8Cons lee.

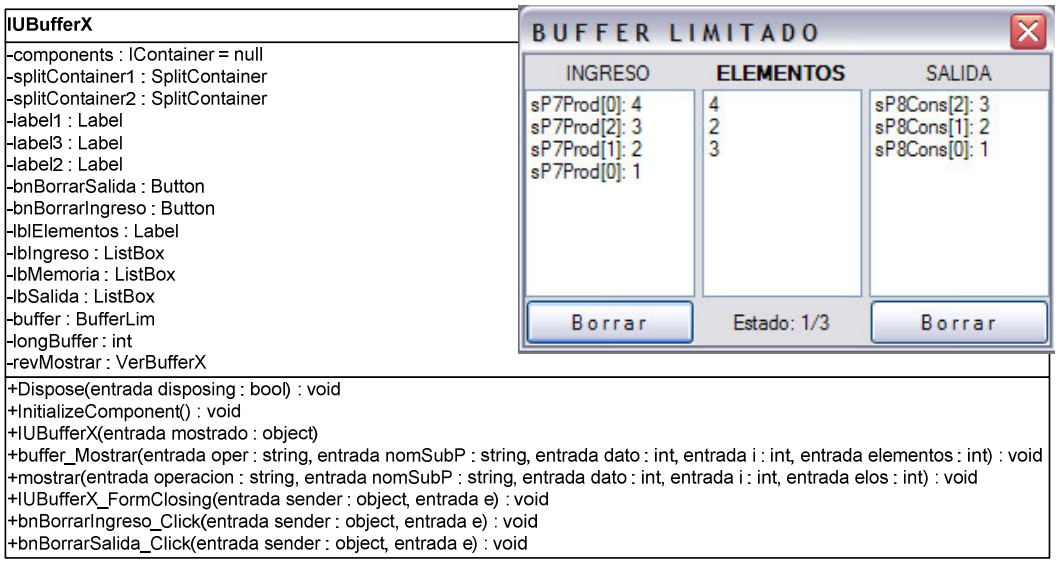


Figura 5.17 Interfaz de búfer de un número limitado de memorias

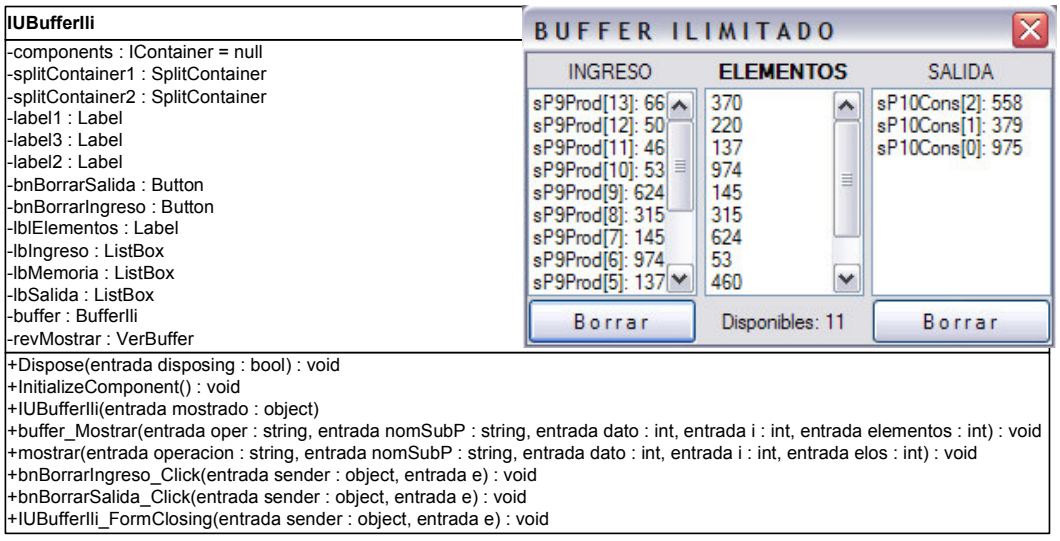


Figura 5.18 Interfaz de búfer sin limitación de memorias

Interfaz de búfer sin limitación de número de memorias.

El formulario que visualiza el búfer sin limitación de memoria y su servicio IUBuffeIli se presentan en la figura 5.18. La clase emplea programación asíncrona. El formulario de la figura muestra búfer sin límite, en el cual el subproceso sP9Prod escribe y del cual el subproceso sP10Cons lee.

5.3.7 TAREAS ESPECÍFICAS.

Adicionalmente a los paquetes que constituyen la propia arquitectura del GASP, el paquete de tareas específicas contiene cuatro subprocesos que facilitan a este gestor la realización de los servicios que presta. El desarrollo de programas (o segmentos) que se convierten en subprocesos se presenta en el siguiente capítulo. En esta parte solo se enuncia las responsabilidades de cada uno de estos subprocesos.

- **spDespertar**, subproceso con 255 como prioridad (máxima prioridad software y hardware) que atiende las interrupciones del temporizador de intervalo programable (TIP). Antes de entrar en bucle habilita su bit de interrupción. En el bucle: recibe un mensaje de interrupción en BuzonIRQ0, captura el semáforo, procesa la lista de subprocesos dormidos y libera el semáforo. Durante el proceso de la lista de dormidos: si hay subprocesos dormidos, decrementa la cuota a dormir del primer subproceso. Si el decremento dio como resultado cero, pasa todos los subprocesos con cuota por dormir cero son pasados de la lista dormidos a la lista listos.
- **spSiempreListo**, subproceso con prioridad 0, que siempre está listo para ejecutarse.
- **spMjeIn**, subproceso con prioridad 225 que recibe entre sus parámetros el mensaje y las veces que el mensaje debe ser insertado en un buzón accesible por plantilla. El subproceso realiza la inserción en la cola de mensajes del buzón empleando el servicio Enviar.
- **spMjeOut**, subproceso con prioridad 225 que recibe entre sus parámetros el número de mensajes que debe retirar de un buzón accesible por pantalla. El subproceso realiza la extracción de mensajes empleando el servicio Recibir.

5.4 INTERFAZ DE INTEGRACION DEL GASP.

La interfaz o entorno de integración constituye el marco, donde las interfaces gráficas del gestor de subprocesos (IGASP) y las interfaces gráficas de los propios subprocesos gestionados adquieren vida. El entorno brinda al usuario los medios para invocar y usar dichas interfaces gráficas. Este entorno esta compuesto por un formulario

con una barra de menús, servido por la clase IGI. Por medio de los menús, los usuarios invocan las interfaces, a través de las cuales conducen el GASP y controlan la dinámica de ejecución de los subprocesos gestionados sobre el GASP. La presentación de este entorno se ejecuta en el subproceso principal del PGA. La figura 5.19 muestra el marco con interfaces, tanto del GASP como de los subprocesos gestionados, y la figura 5.20 especifica completamente la clase IGI. La figura 5.20, también, muestra las relaciones de la clase IGI con otras clases que los subprocesos gestionados comparten.

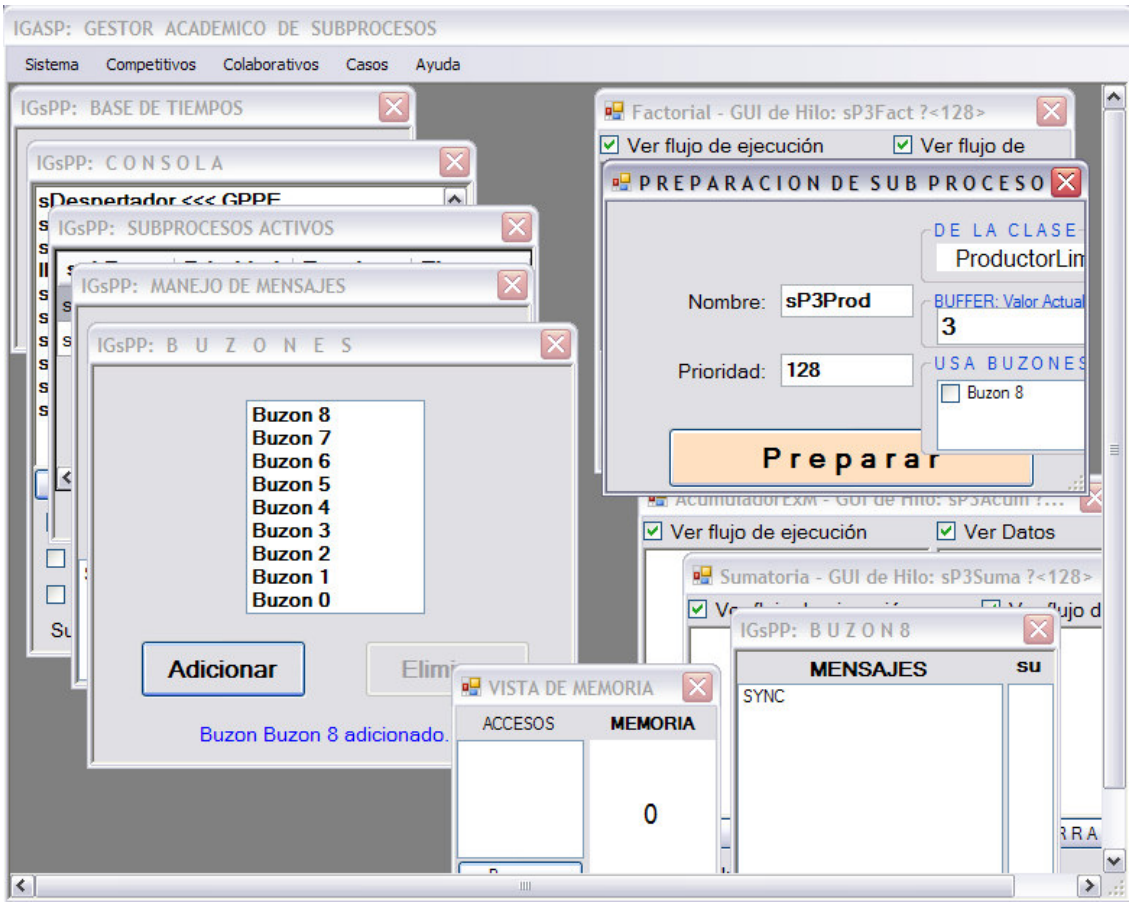


Figura 5.19 Interfaz de integración del GASP - IGASP

Concordante con el párrafo 4.3.1 del capítulo anterior, el primer menú y los cinco formularios internos del lado izquierdo del la figura 5.19 corresponden a las interfaces del GASP. Los diseños de estas interfaces fueron parte del diseño del paquete IGestor. Por medio de estas interfaces el usuario interesado configura el GASP, aprecia su evolución y controla su funcionamiento.

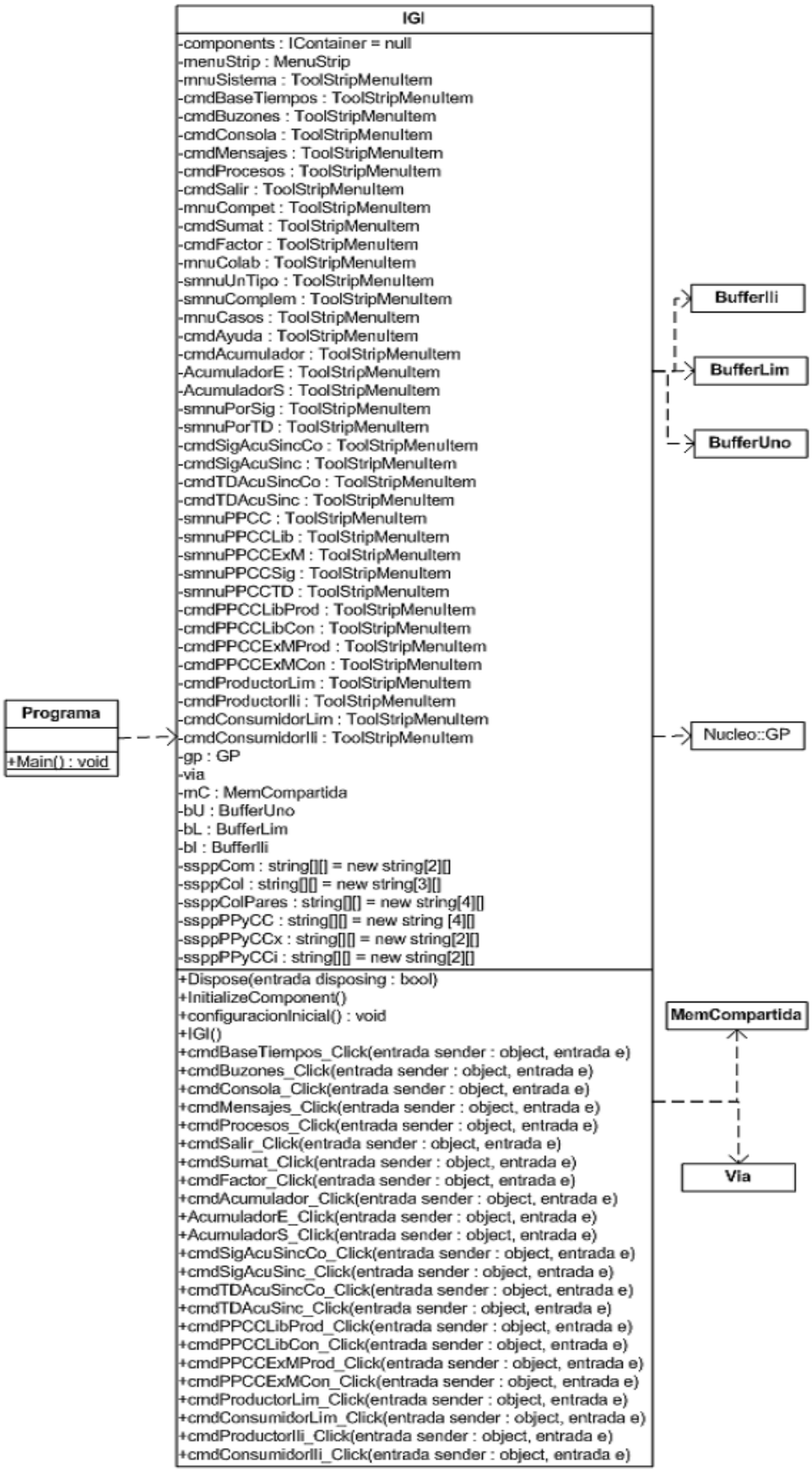


Figura 5.20 Clase de servicio de interfaz IGASP

Las interfaces internas del lado derecho del formulario marco y sus respectivos menús de la figura 5.19 son usados en la gestión de subprocesos dentro de la configuración vigente del GASP. Entre estas interfaces internas, se encuentran tanto las interfaces facilitadas por el GASP y compartidas por los subprocesos gestionados, como las interfaces de los propios subprocesos gestionados. Las interfaces facilitadas por el GASP para compartir incluyen: interfaz de buzón, interfaz de memoria compartida, interfaz de búfer de una memoria, interfaz de búfer de memorias limitadas, interfaz de búfer sin limitación de memorias e interfaz de preparación de subprocesos. Los diseños de las cinco primeras interfaces se incluyeron en el diseño del paquete IGestor y el diseño de la última se presenta en el siguiente párrafo. Las interfaces de los subprocesos gestionados son parte del siguiente capítulo.

Como clase de servicio de interfaz principal de usuario, tal como se muestra en la figura 5.20, IGI incluye tanto atributos para referenciar objetos compartidos por subprocesos gestionados y definir menús, como métodos para preparar objetos compartidos y atender comandos de menús.

El principal objeto compartido es el núcleo del GASP. El objeto Via sirve de puente para que la memoria compartida, los búferes, los buzones y otros objetos sean compartidos entre los subprocesos. Para los subprocesos interactivos incorporados, determinados parámetros están predefinidos en arreglos de cadenas referenciados por atributos del IGI. Los parámetros usados por un subproceso interactivo son seleccionados y completados por el usuario por medio de la interfaz preparación de subprocesos.

El constructor, con la ayuda de los métodos InitializeComponent y configuracionInicial, realiza la configuración inicial de la interfaz de integración y del GASP. Los otros métodos de IGI atienden a los eventos generados por el usuario por medio de los menús e invocan, pasando los respectivos parámetros, a interfaces del GASP o a la interfaz de preparación de subproceso gestionado.

En el diagrama de la figura 5.20 se incluye la clase Programa que define Main, el primer método en ejecutarse en una aplicación desarrollada en C#.

5.4.1 INTERFAZ DE PREPARACION DE SUBPROCESO.

Cada comando de los menús: competitivos, colaborativos y casos, invoca esta interfaz. La interfaz permite al usuario seleccionar e ingresar los parámetros que usa un

subproceso gestionado interactivo y los captura en el objeto Via. El comando, que invocó a la interfaz, adiciona al objeto Via los restantes objetos requeridos por el subproceso y termina creando la interfaz de gestión del subproceso e interfaces asociadas (vistas: memoria, búferes y/o buzones). En la figura 5.21 se presenta esta interfaz y la especificación completa de su servicio IUpreIUsp.

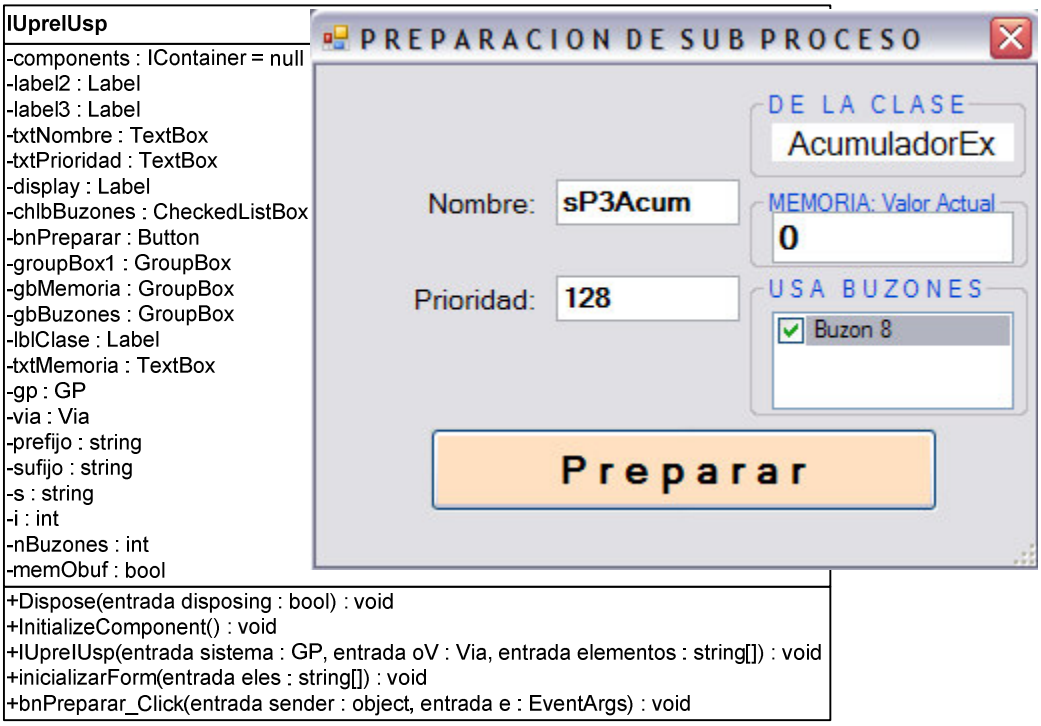


Figura 5.21 Interfaz de preparación de parámetros de subproceso gestionado interactivo

Los atributos de la clase IUpreIUsp apuntan a los controles del formulario, definen las variables locales de trabajo (prefijo, sufijo, s, i, nBuzones, memObuf) y retienen las referencias al núcleo del GASP y al objeto Via. Con el núcleo del GASP (sistema), el objeto Via (oV) y los datos predefinidos de subproceso (elementos), recibidos como parámetros de entrada, el constructor realiza la configuración inicial del formulario con la ayuda del método inicializarForm. El botón de comando bnPreparar se encarga de invocar al método bnPreparar_Click que coloca los datos aceptados en el objeto Via.

5.5 ARQUITECTURA DE IMPLEMENTACION Y DESPLIEGUE.

El GASP y los subprocesos que gestiona se integran en una aplicación desktop, que se construye en una solución (SisGASP) de vs.net con lenguaje C# y que se instala

en un computador con .NET. Tal como se muestra en la figura 5.22, la aplicación SisGASP está compuesta de tres subsistemas, implementados en sus respectivos proyectos de programación: GASP, IGASP y subProcesos. Dentro de cada proyecto, las clases – diseño se implementan como archivos de código fuente en lenguaje C# (componentes de producción) organizados en paquetes, a partir de los cuales se construye un componente desplegable (archivo con código en lenguaje intermedio). Los componentes desplegables del SisGASP son: GASP.DLL, subProcesos.DLL e IGASP.EXE.

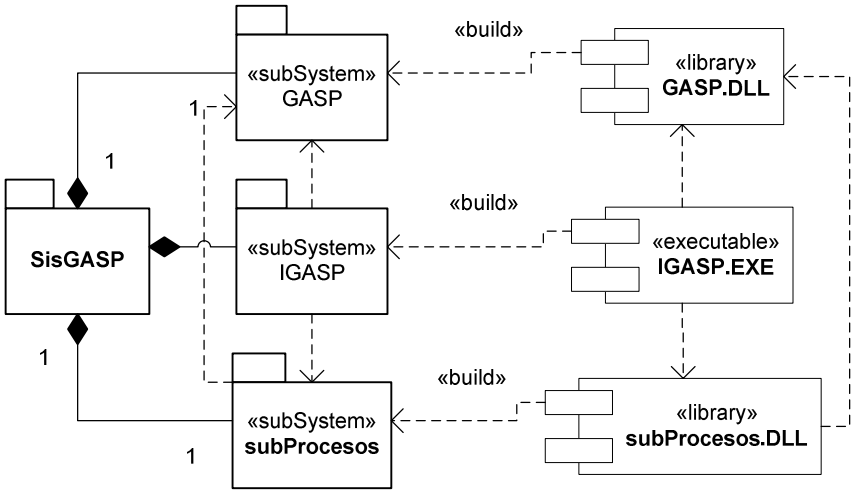


Figura 5.22 Arquitectura contextual de la implementación del GASP

El GASP.DLL es un componente tipo librería de clases que implementa las clases del GASP, tanto de su capa de aplicación como de su capa de presentación. El subProcesos.DLL implementa los subprocesos gestionados y sus interfaces, que se ven en el siguiente capítulo. IGASP.EXE es un componente desplegable del tipo aplicación de ventanas (Windows Application) que implementa las clases de la interfaz de integración (IGASP).

5.6 COMPONENTES DEL GASP.

Los componentes de producción del GASP se organizan en paquetes de implementación mostrados en la figura 5.23. Existe una traza de uno a uno entre los paquetes de implementación y los paquetes de clases – diseño del GASP.

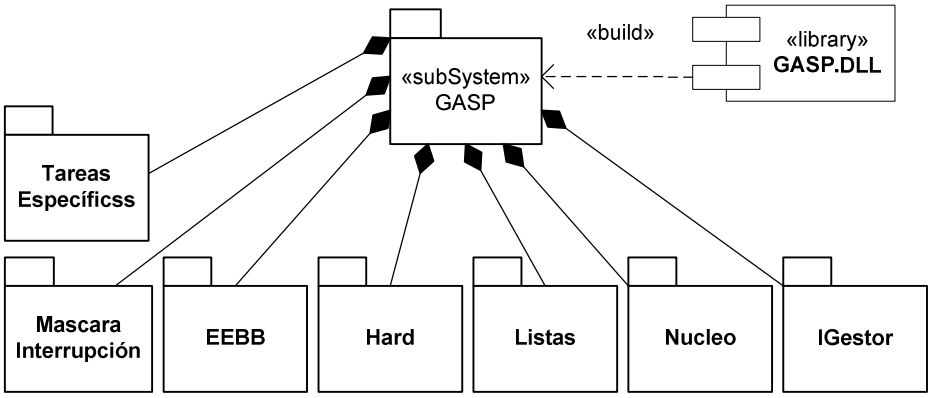


Figura 5.23 Paquetes de componentes de producción del GASP

En los siguientes párrafos se presentan los componentes de cada paquete, donde cada componente (archivo con código en lenguaje C#) suele implementar parcialmente una clase de implementación. El código en lenguaje C# de cada archivo fuente se presenta en Fuentes de GASP de anexos; a excepción de los voluminosos archivos IUxxx.Designer.cs que poco aportan a las interfaces especificadas en el diseño.

5.6.1 MASCARA DE INTERRUPCIONES.

El paquete Máscara Interrupción contiene la clase de implementación MascaraInt que se codifica en los archivos MascaraInt1.cs y MascaraInt2.cs, tal como se muestra en la figura 5.24.

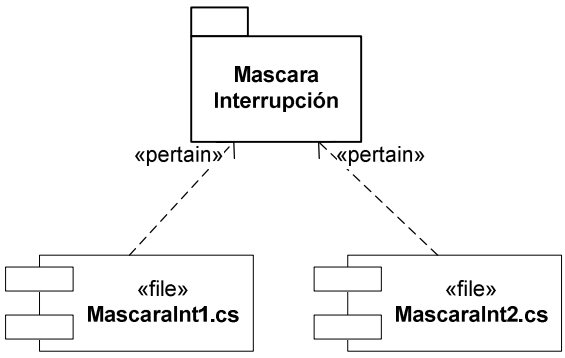


Figura 5.24 Componentes en Mascara Interrupción

5.6.2 EMULADOR HARDWARE.

El paquete Hard contiene las clases de implementación: BaseTiempos, CIP, Interrupción, MemCompartida, BufferUno, BufferLim y BufferIli. Cada una de estas clases se codifica en sus respectivos archivos, tal como se aprecia en la figura 5.25.

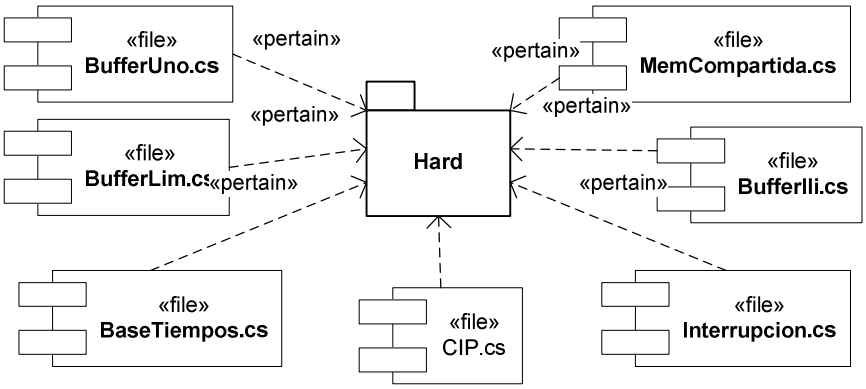


Figura 5.25 Componentes en Hard

5.6.3 ELEMENTOS BASICOS.

El paquete EEBB contiene las clases de implementación: BCP, Mensaje y Buzón. Tal como se aprecia en la figura 26, BCP se codifica en cuatro archivos, Buzón se codifica en seis archivos y Mensaje se codifica en dos archivos.

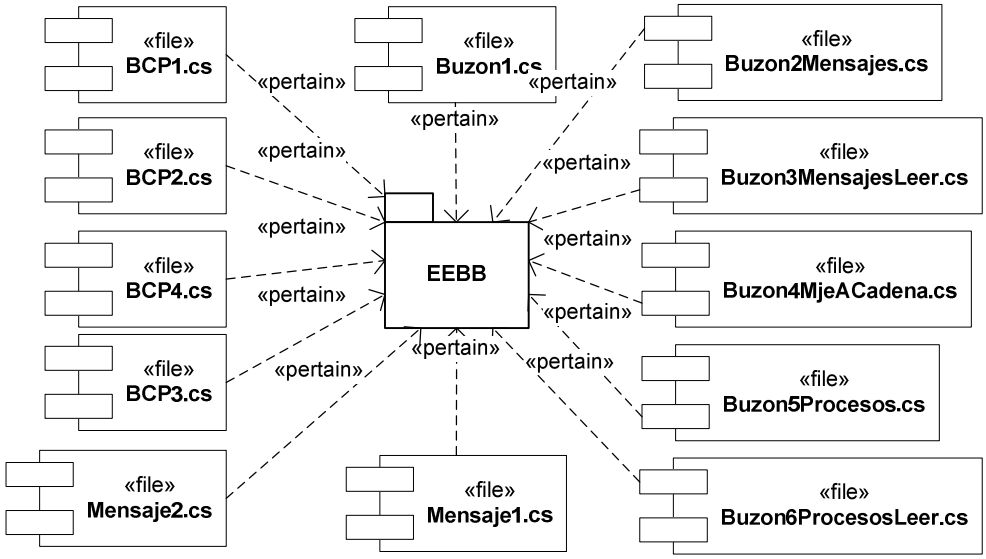


Figura 5.26 Componentes en EEBB

5.6.4 LISTAS DE ELEMENTOS BASICOS.

El paquete listas contiene la clase listas. Tal como se aprecia en la figura 27, esta clase de implementación se codifica en siete archivos.

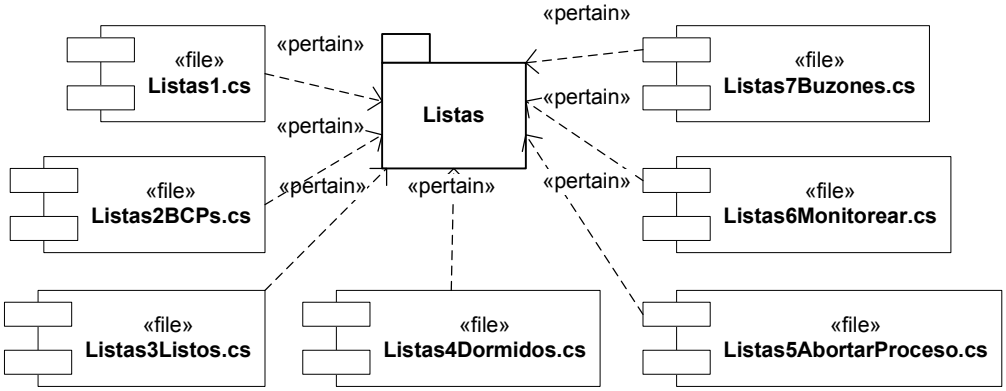


Figura 5.27 Componentes en Listas

5.6.5 NUCLEO.

El paquete núcleo contiene la clase de implementación GP, que se codifica en dieciocho archivos que se presentan en la figura 5.28.

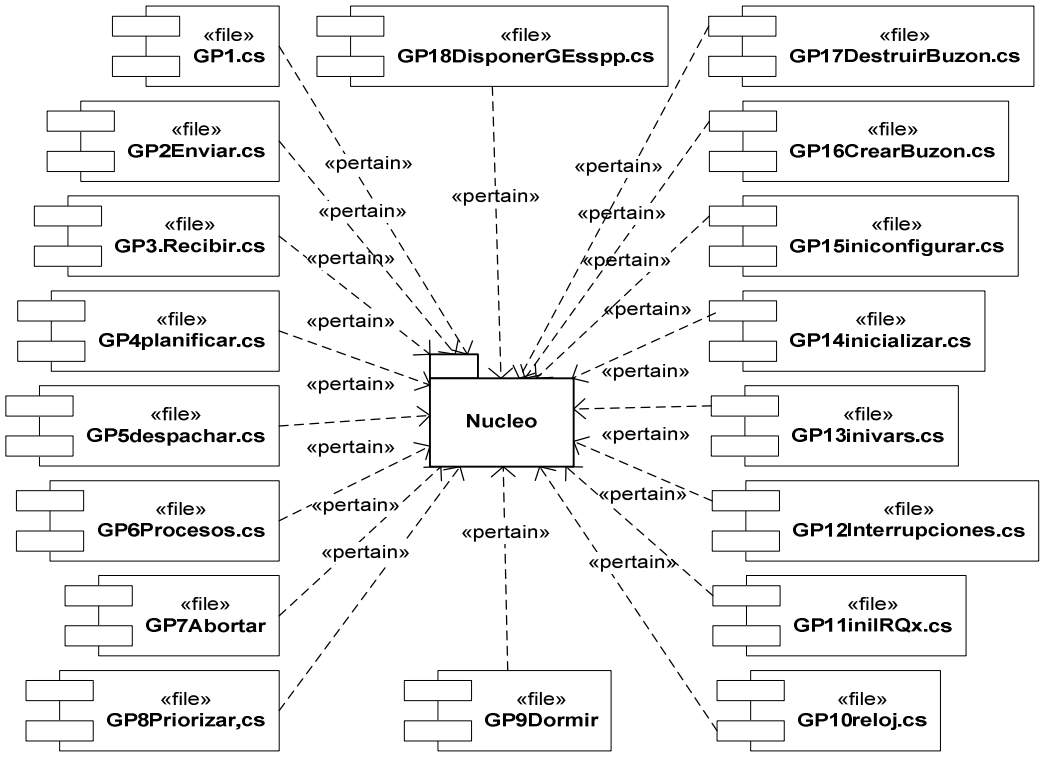


Figura 5.28 Componentes en Núcleo

5.6.6 INTERFAZ DEL GASP.

Los componentes de producción de la interfaz del GASP se encuentran en el paquete IGestor. Este paquete contiene las clases de implementación: IUBaseTiempos, IUBuzones, IUConsola, IUMensajes, IUProcesos, IUBuzon, IUMemoria, IUBufferUno,

IUBufferIli e IUBufferX. Cada una de estas clases se codifica en dos archivos, tal como se muestra en la figura 5.29.

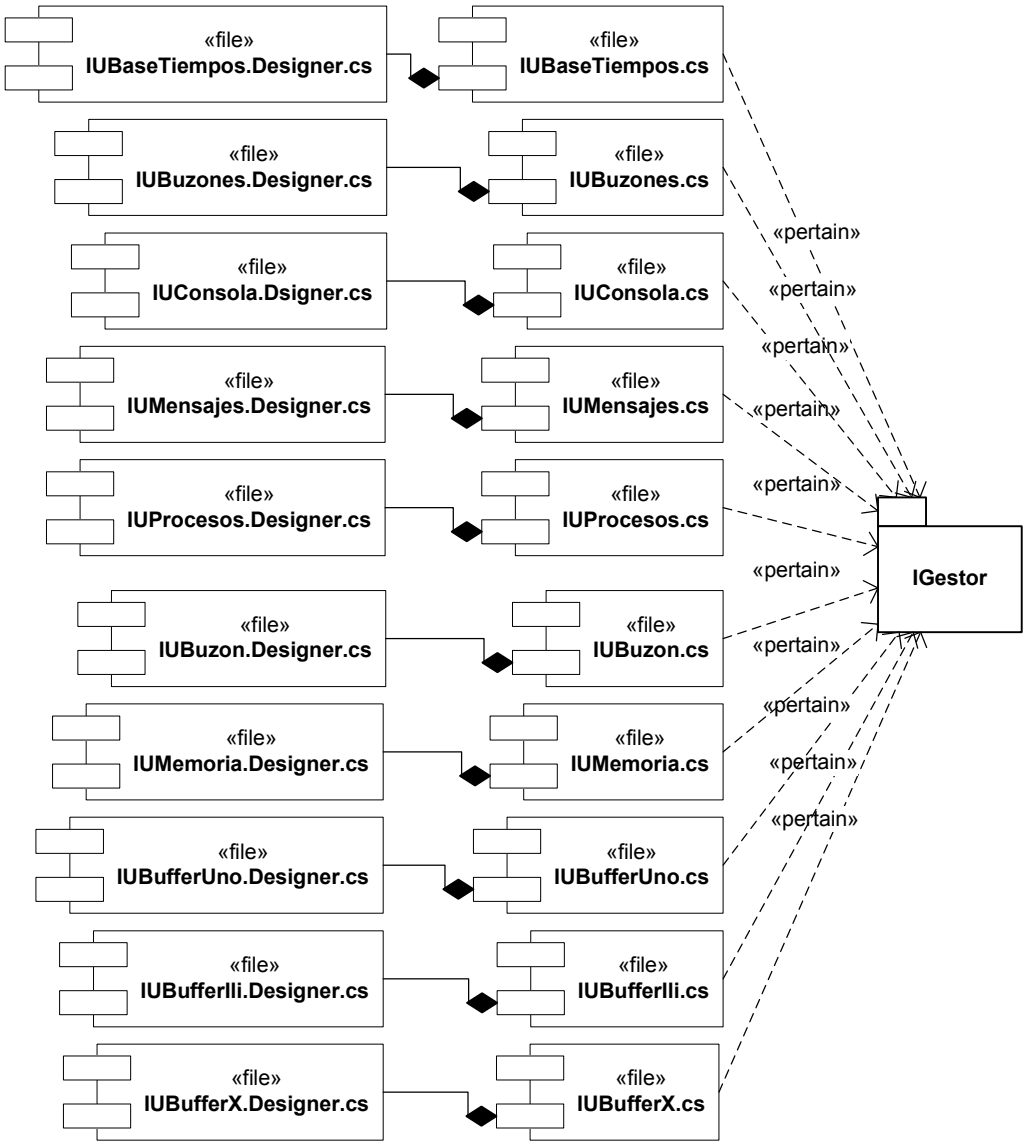


Figura 5.29 Componentes en IGestor

5.6.7 TAREAS ESPECÍFICAS.

El paquete tareas específicas contiene las siguientes clases de implementación: Tarea, Via, pDespertador, pSiempreListo, pMjeIngresar y pMjeRetirar. Tal como se observa en la figura 5.30, cada clase se codifica en su archivo.

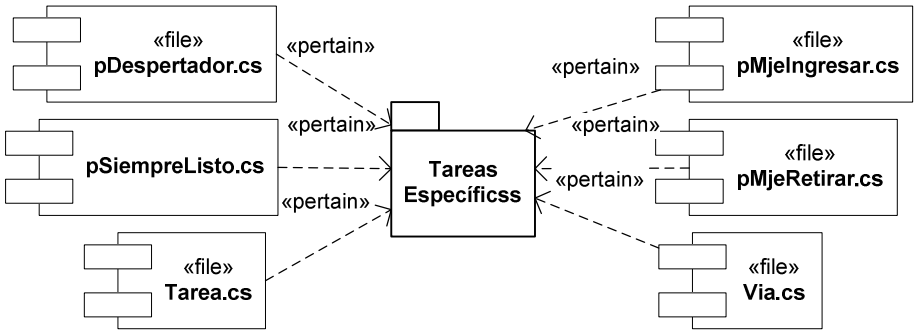


Figura 5.30 Componentes en Tareas Específicas

5.7 COMPONENTES DE LA INTERFAZ DE INTEGRACION.

La integración del GASP con los subprocesos gestionados se realiza por medio del subsistema interfaz de integración (IGASP). Este subsistema contiene las clases de implementación programa, IGI y IUpreIUsp, a partir de las cuales se construye IGASP.EXE. Tal como aparece en la figura 5.31, cada una de las últimas dos clases se codifican en dos archivos. El código en lenguaje C# de cada archivo fuente se presenta en Fuentes de IGASP del anexo, sin considerar los archivos xxx.Designer.cs que quedan exonerados por las respectivas interfaces especificadas en el diseño

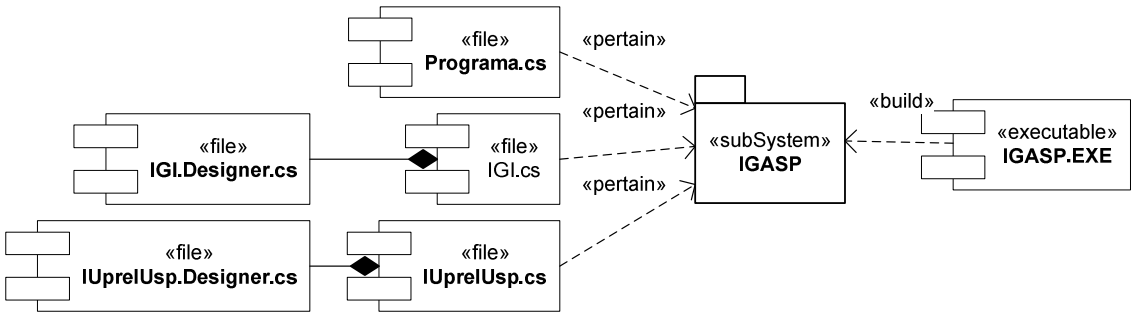


Figura 5.31 Componentes del subsistema IGASP

En el siguiente capítulo se describen los subprocesos gestionados y se presentan ejemplos de estos subprocesos como casos de prueba del GASP.

Capitulo VI:
SUBPROCESOS GESTIONADOS EN EL GASP.

El gestor académico de subprocesos (GASP) y la interfaz de integración (IGASP), diseñados e implementados en el capítulo anterior, soportan la construcción de subprocesos gestionados en el entorno vs.net 2005 (y posteriores) con lenguaje C# y la gestión de su ejecución referencial a velocidades controladas por el usuario. El subproceso gestionado (SPG) es un motor de ejecución que realiza una tarea de cómputo con ejecución referencial al lenguaje C# y con velocidad de ejecución controlada por el usuario. Estas tres propiedades del SPG explícitamente se especifican en el código fuente en lenguaje C#.

Los párrafos 5.2.1 y 5.2.2 del capítulo anterior presentaron la arquitectura de del SPG y el párrafo 5.5, del mismo capítulo, especificó que todos los subprocesos gestionados se implementan en un subsistema (subProcesos) de la aplicación desktop SisGASP. La aplicación SisGASP se ejecuta en un proceso sobre la plataforma .NET que se ha denominado proceso del gestor académico (PGA). El PGA contiene un subproceso principal (no gestionado) y cero o más subprocesos gestionados. La estructura del SPG, incluyendo su arquitectura, es materia del presente capítulo, acompañada del resumen de su proceso de construcción y de los ejemplos considerados como casos demostrativos.

6.1 ESTRUCTURA DEL SPG.

Concordante con el objetivo del presente trabajo, lograr una herramienta sencilla para emplearse en el proceso de enseñanza – aprendizaje de subprocesos y procesos de sistemas operativos, el diseño del subproceso gestionado se mantiene simple: tanto a nivel arquitectónico como a nivel de clases y objetos en sus capas.

6.1.1 ARQUITECTURA DEL SUBPROCESO.

Independiente de la tarea de cómputo que realice el subproceso gestionado, su arquitectura está compuesta de las capas de presentación y de aplicación. Tal como se muestra en la figura 6.1, la interfaz gráfica de usuario final del subproceso (ISP) constituye la primera capa y la lógica de aplicación del subproceso (SP) constituye la

segunda. Los subprocesos gestionados que no interactúan con el usuario final no requieren de la capa de presentación, como es el caso de los subprocesos reactivos y trabajadores. Los subprocesos interactivos requieren de las dos capas. El diseño básico de clases y objetos de ambas capas se presentan en los siguientes párrafos.

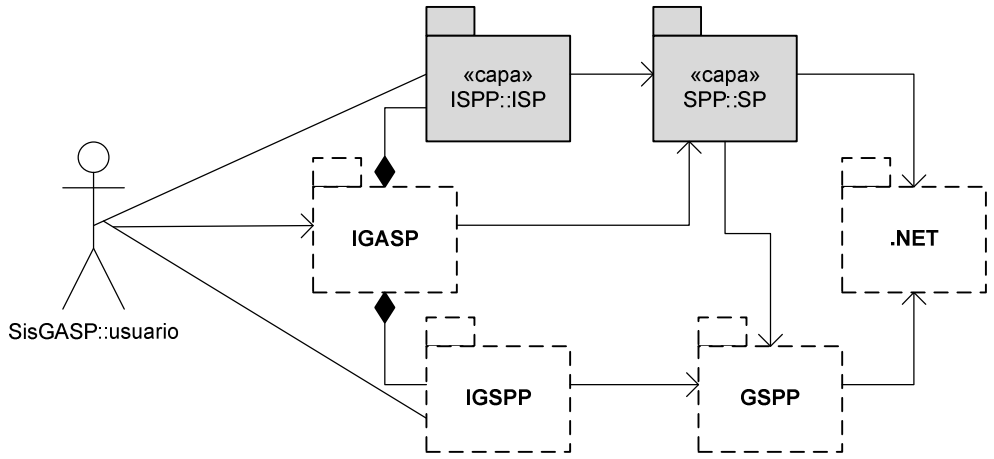


Figura 6.1 Arquitectura en capas del subproceso gestionado en su contexto

6.1.2 INTERFAZ DE USUARIO DEL SUBPROCESO.

La interfaz de usuario final del subproceso gestionado interactivo está constituida fundamentalmente por un formulario y la clase que lo administra. La interfaz recibe el núcleo del GASP y un objeto Via. El objeto Via porta las características elegidas para el subproceso que gestiona la interfaz y los elementos del GASP requeridos por este subproceso. Una interfaz contiene elementos comunes a todos los subprocesos y elementos específicos del subproceso que gestiona.

En la figura 6.2 se presenta el diseño básico de la interfaz de un SPG, en términos de su apariencia, sus elementos (comunes y algunos específicos) y su comportamiento:

Los **elementos comunes** de la interfaz son los siguientes:

- Barra de título con la identificación del subproceso y su prioridad,
- Panel para mostrar el flujo de ejecución referencial del subproceso en lenguaje C# con sus controles asociados para habilitación/deshabilitación y borrado,
- Panel para mostrar los resultados (datos) intermedios durante la ejecución del subproceso con sus controles asociados para habilitación/deshabilitación y borrado,
- Botón de comando para crear el subproceso y
- Control de cierre de formulario para abortar el subproceso.

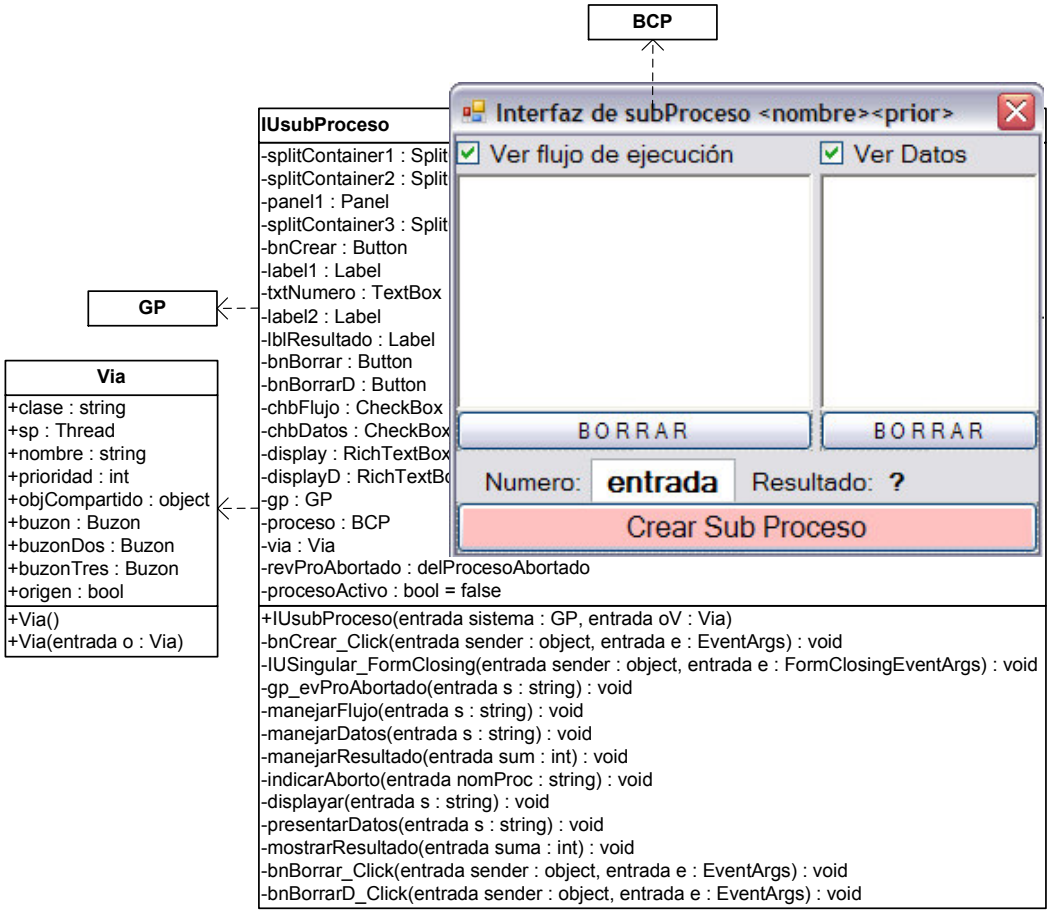


Figura 6.2 Interfaz de usuario final del subproceso gestionado

Los **elementos específicos** de la interfaz que se muestran en la figura 6.2 corresponden a los controles para número de entrada y resultado final, así como para la definición de la apariencia. Las vistas de buzones, de memoria compartida, y de búferes suelen extender las interfaces de los subprocesos gestionados colaborativos. Las interfaces de estos tipos de subprocesos incluyen botones de comando para visualizar/ocultar las mencionadas vistas.

El comportamiento de la interfaz es conducido por eventos, procesados por sus métodos. Determinados métodos de la interfaz se ejecutan en el subproceso principal del PGA, los demás - en el subproceso gestionado.

Los **eventos del SPG** (y de otros formularios) de interés para la interfaz se manejan por programación asíncrona. El procesamiento de cada uno de este tipo de eventos involucra dos métodos: el primero que se ejecuta sincrónicamente en el SPG y otro que se ejecuta asincrónicamente en el subproceso principal, como efecto del primero. La figura 6.2 incluye los siguientes eventos de este tipo:

- Flujo de ejecución presente, que involucra los métodos: manejarFlujo y displayar.

- Resultado intermedio obtenido, que involucra los métodos: manejarDatos y presentarDatos.
- Resultado final calculado, que involucra los métodos: manejarResultado y presentarResultado,
- Subproceso gestionado abortado, que involucra los métodos: gp_evProAbortado e indicaAborto.

Los métodos que procesan **eventos propios de la interfaz**, así como otros métodos, se ejecutan sincrónicamente en el subproceso principal del PGA. La figura 6.2 especifica los siguientes métodos de este tipo: constructor, bnCrear_Click, IUSingular_FormClosing, bnBorrar_Click y bnBorrarD-Click.

El método bnCrear_Click se encarga de crear el subproceso y registrarlo en el GASP para su gestión, así como inscribir los métodos de la interfaz a los eventos del subproceso.

6.1.3 LOGICA DE APLICACIÓN DEL SUBPROCESO.

La organización básica de clases y objetos dentro de la capa lógica de aplicación del subproceso gestionado se presenta en la figura 6.3. La clase base (Tarea) facilita el uso de los servicios y los elementos del GASP. La clase derivada (sPTareaEspecífica) define la tarea de cómputo específica del SPG. La naturaleza del cómputo puede imponer clases adicionales.

El método Ejecutar, que se define vacío en Tarea, en el subproceso debería contener el código principal o inicial. En la figura 6.3, todo el código que define la tarea de cómputo del subproceso gestionado esta contenido dentro de la redefinición del método Ejecutar.

Junto a la clase Tarea, el GASP facilita un conjunto de delegados que los subprocesos gestionados pueden incluir para publicar sus eventos a su interfaz y a interfaces del GASP, entre los que se encuentran el que aparece en la figura 6.3 y los siguientes: `public delegate void IndicaDatoString(string s)`, `public delegate void IndicaDatoObject(object o)`, `public delegate void IndicaCodigo(string s)` y `public delegate void IndicaSignal()`.

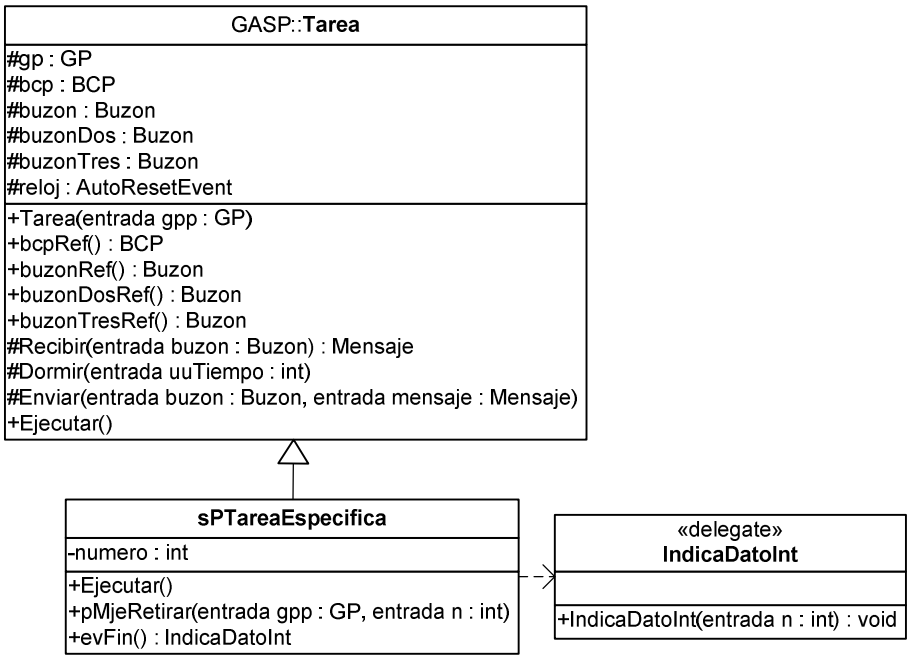


Figura 6.3 Lógica de aplicación de subproceso gestionado

6.2 CONSTRUCCION DEL SPG.

Conocidos los requisitos de la tarea de cómputo a realizarse en un subproceso gestionado (SPG), expresados en un enunciado de especificaciones u en otro medio, el proceso de construcción fundamentalmente consiste en diseñar la estructura del SPG, implementarlo y probarlo. La tarea de cómputo elegida es la siguiente: calcular y presentar la sumatoria de los números naturales, hasta un número (inclusive) dado por el usuario. Este párrafo presenta el diseño y la implementación del SPG que realiza esta tarea. La implementación se presenta en las perspectivas del modelo de componentes de producción y la programación en lenguaje C#. La prueba de este SPG se presenta como un caso demostrativo de SPG en el GASP.

6.2.1 DISEÑO DE LA ESTRUCTURA.

El subproceso gestionado sumatoria está estructurado en los clasificadores que se muestran en la figura 6.4. El GASP facilita la clase Tarea y los delegados IndicaCodigo, IndicaDatoInt e IndicaDatoString. Las clases específicas del subproceso son IUSingular y Sumatoria. IUSingular gestiona la interfaz de usuario del subproceso y Sumatoria realiza la tarea de cómputo (lógica de aplicación).

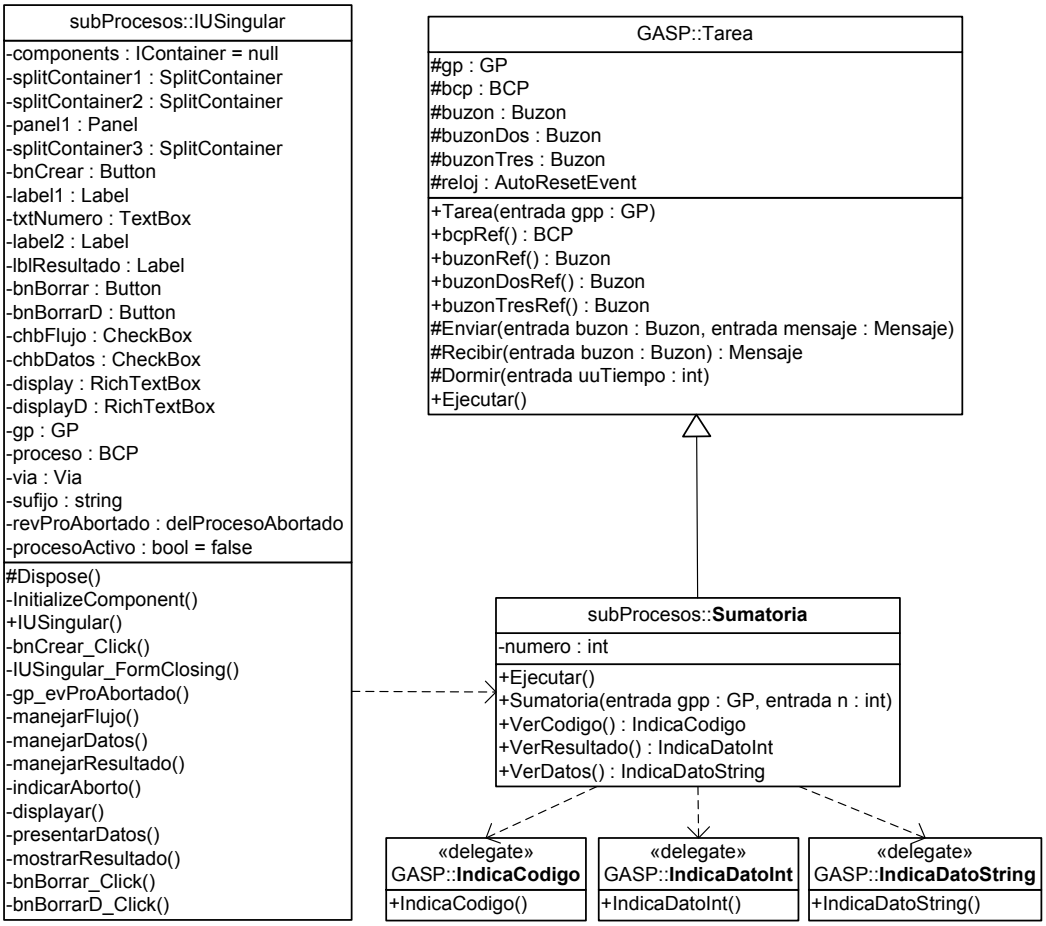


Figura 6.4 Estructura del SPG Sumatoria

La estructura y el comportamiento de la clase IUSingular son similares a los presentados en el párrafo 6.1.2.

En la clase Sumatoria, el método sobrescrito Ejecutar realiza la tarea de cómputo empleando el número ingresado por el usuario y el contexto del GASP, recibidos por el constructor. La clase Tarea pone a disposición del método Ejecutar los elementos del GASP, entre ellos la velocidad controlada (objeto reloj). Los delegados definen los eventos que el método Ejecutar publica para referenciar el avance de su ejecución, presentar los resultados.

6.2.2 COMPONENTES DE PRODUCCION.

Como se especificó al inicio del capítulo, todos los subprocesos gestionados interactivos se implementan en el subsistema subProcesos, el cual se presenta en la

figura 6.5. Este subsistema tiene dos paquetes de implementación: ISPP para las capas de presentación de los subprocesos y SPP para las capas de aplicación.

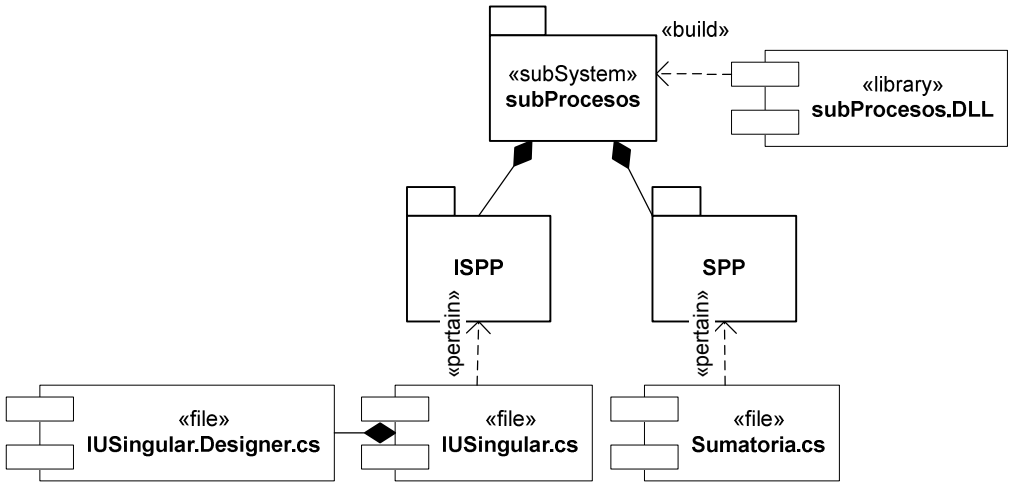


Figura 6.5 Paquetes del subsistema subProcesos y componentes del SPG Sumatoria

La figura 6.5 también incluye los componentes de producción del SPG Sumatoria, ubicados en sus respectivas capas: dos archivos para la clase IUSingular en ISPP y un archivo para la clase Sumatoria en SPP. Los otros componentes del subproceso ya están implementados en el subsistema GASP.

6.2.3 CODIGO FUENTE EN C#.

El código fuente de los subprocesos gestionados se incluye en Fuentes de subProcesos gestionados de anexos, excluyendo los grandes archivos IUxxx.Designer.cs que dibujan las respectivas interfaces presentadas mas adelante, en Casos demostrativos de SPGs en el GASP. El presente párrafo presenta el código de las clases Tarea, Sumatoria y IUSingular, de ésta última en forma parcial.

La clase base **Tarea se** implementa en el subsistema GASP y, tal como se aprecia en la figura 6.6 (líneas 07 a 51), recibe los objetos del GASP a través de su constructor y sus propiedades, y los pone a disposición de sus clases derivadas. En las líneas 53 a 70, empleando los servicios del GASP, esta clase define los métodos Recibir, Enviar y Dormir para las clases que lo heredan. Finalmente, en la línea 72, la clase Tarea define el método virtual Ejecutar.

```
01 using System;
02 using System.Threading;
03
04 namespace GASP
05 {
06  public class Tarea
07  {
08   protected GP gp; //nucleo de gestor de subprocesos
09   protected BCP bcp; //descriptor del subproceso
10   protected Buzon buzon; //buzones de mensajes
11   protected Buzon buzonDos;
12   protected Buzon buzonTres;
13   protected AutoResetEvent reloj; //reloj de ejecucion
14
15   public Tarea(GP gpp) //constructor
16   {
17    gp = gpp;
18   }
19
20    public BCP bcpRef //establece referencia a descriptor de SPG
21    {
22     set
23     {
24      bcp = value;
25      reloj = bcp.Reloj; //referencia a reloj de GASP
26     }
27    }
28
29    public Buzon buzonRef //establece referencia a buzon
30    {
31     set
32     {
33      buzon = value;
34     }
35    }
36
37    public Buzon buzonDosRef //establece referencia a buzon
38    {
39     set
40     {
41      buzonDos = value;
42     }
43    }
44
```

Figura 6.6 Código fuente de la clase Tarea del subsistema GASP

```
45   public Buzon buzonTresRef //establece referencia a buzon
46   {
47     set
48     {
49       buzonTres = value;
50     }
51   }
52
53   protected void Enviar(Buzon buzon, Mensaje mensaje)
54   {
55     gp.Enviar(buzon, mensaje); //servicio de GASP
56     reloj.WaitOne();
57   }
58
59   protected Mensaje Recibir(Buzon buzon)
60   {
61     gp.Recibir(buzon); //servicio de GASP
62     reloj.WaitOne();
63     return bcp.Mensaje;
64   }
65
66   protected void Dormir(int uuTiempo)
67   {
68     gp.Dormir(uuTiempo); //servicio de GASP
69     reloj.WaitOne();
70   }
71
72   public virtual void Ejecutar(){}
73 }
74 }
```

Figura 6.6 Código fuente de la clase Tarea del subsistema GASP (continuación)

La clase **Sumatoria** se presenta en dos versiones: la primera en la figura 6.7, en un programa convencional y ejecutable en modo consola, y la segunda en la figura 6.8, como la capa de aplicación del SPG. El código de la primera versión es simple, por lo que no requiere explicación, y su propósito es facilitar la obtención del código de la segunda versión.

En el proceso de conversión de la primera versión a la segunda de la clase Sumatoria, tal como aparece en la figura 6.8: se introduce la clase base Tarea, se reemplaza el método Main por el constructor y la interfaz del SPG (que invoca al método Ejecutar), se conservan las demás instrucciones (líneas 20 a 90), se adiciona las declaraciones de eventos (líneas 25 a 27) y se insertan instrucciones para controlar la velocidad de ejecución y la referencia de esta ejecución en el lenguaje C# hacia su interfaz (líneas 39 a 95).

```
01 using System;

10 public class Sumatoria
11 {
15  static void Main(string[] args)
16  {
17    numero = 5;
18    Ejecutar();
19  }

20  static int numero;

30  public static void Ejecutar()
31  {
40    int n = 0;
50    int acumulado = 0;
60    while (n < numero)
61    {
70      n++;
80      acumulado += n;
89    }
90    Console.Write(acumulado);
97  }
98 }
```

Figura 6.7 Clase Sumatoria en programa convencional

Dentro del algoritmo de cómputo (método Ejecutar), en promedio se emplean tres instrucciones adicionales para gestionar cada instrucción (o grupo): una para publicar la propia instrucción, otra para publicar el resultado intermedio y una tercera para marcar el tiempo. Por ejemplo, la instrucción 70 (*n++;*) en el programa 6.3 va precedida de la instrucción 69 (*VerCodigo("n++\n");*) que lo publica y va seguida de las instrucciones 71 (*VerDatos(string.Format("n={0}\n", n));*) que publica el dato afectado y 72 (*reloj.WaitOne();*) que marca el tiempo. De manera similar cada instrucción original del método Ejecutar está acompañada de tres instrucciones adicionales. La instrucción 95 finaliza el SPG en el GASP.

La interfaz implementada por la clase **IUSingular** se emplea para gestionar subprocesos no colaborativos, dentro de los cuales se encuentra el SPG Sumatoria. En la figura 6.9 se presenta un extracto de código fuente del método bnCrear_Clock, relacionado con la gestión del SPG Sumatoria.

```
01 using System;
02 using GASP;

05 namespace subProcesos
06 {
10  public class Sumatoria : Tarea
11  {
15    public Sumatoria(GP gpp, int n) : base(gpp) //constructor
16    {
17      numero = n;
19    }

20    int numero;
    //declaracion de eventos
25    public event IndicaCodigo VerCodigo;
26    public event IndicaDatoInt VerResultado;
27    public event IndicaDatoString VerDatos;

30    public override void Ejecutar() //algoritmo de computo
31    {
39      VerCodigo("n = 0\n"); //publicar flujo de ejecucion
40      int n = 0;
41      reloj.WaitOne(); //esperar
49      VerCodigo("acumulado = 0\n"); //publicar flujo de ejecucion
50      int acumulado = 0;
51      VerDatos(string.Format("S = {0}, n={1}\n",acumulado, n));//fDD
52      reloj.WaitOne();
59      VerCodigo("while (n < numero)\n");
60      while (n < numero)
61      {
62        VerCodigo("(n < numero)\n");
63        reloj.WaitOne();
69        VerCodigo("n++\n");
70        n++;
71        VerDatos(string.Format("n={0}\n", n));
72        reloj.WaitOne();
79        VerCodigo("acumulado += n\n");
80        acumulado += n;
81        VerDatos(string.Format("S = {0}\n", acumulado));//pubFlujoDD
82        reloj.WaitOne();
83      }
84      VerDatos(string.Format("S = {0}\n\n", acumulado));//pubFlujoDD
89      VerCodigo("acumulado =>\n");
90      VerResultado(acumulado); //publicar resultado final
94      VerCodigo("gp.Fin()\n");
95      gp.Fin(); //finalizar subproceso gestionado en GASP
97    }
98  }
99 }
```

Figura 6.8 Código fuente de la clase Sumatoria del SPG Sumatoria.

```
01 private void bnCrear_Click(object sender, EventArgs e)
02 {
03  bool ver = true;
04  int numero = 0;
05  Sumatoria objSum; //tipo de subProceso
07  Thread spEjecutar; //subProceso
08  try //verificar datos de entrada
09  {
10    numero = int.Parse(txtNumero.Text);
11  }
12  catch
13  {
14    ver = false;
15  }
17
18  if (ver)
19  { //datos de entrada correctos
20    bnCrear.Enabled = false;
21    txtNumero.Enabled = false;
22    lblResultado.Text = "";
23    via.nombre = "sP" + BCP.Todos + sufijo;
24    Text = via.clase + " - GUI de SPG: " + via.nombre + "<" + via.prioridad + ">";
26    if (via.clase == "Sumatoria")
27    { //subproceso del tipo Sumatoria
28      objSum = new Sumatoria(gp, numero); //instancia sumatoria
            //crear subProceso en .NET
29      spEjecutar = new Thread(new ThreadStart(objSum.Ejecutar));
            //crear subProceso en GASP y autoreferenciar
30      objSum.bcpRef = proceso = gp.CrearProceso(spEjecutar,via.prioridad,via.nombre);
            //registrar procesadores de eventos en subProceso
32      objSum.VerCodigo += new IndicaCodigo(this.manejarFlujo);
33      objSum.VerDatos += new IndicaDatoString(this.manejarDatos);
34      objSum.VerResultado += new IndicaDatoInt(this.manejarResultado);
35    }
46    spEjecutar.Name = via.nombre; //asignar nombre a subProceso en .NET
47    spEjecutar.Start(); //arrancar ejecución de subProceso
48
49    gp.evProAbortado += revProAbortado; //registrar procesador de evento en GASP
50    procesoActivo = true;
61  }
62  else
63  { //datos de entrada incorrectos
64    display.AppendText("Ingrese numero correcto...\n");
65    display.ScrollToCaret();
66  }
67 }
```

Figura 6.9 Extracto de código fuente del método bnCrear_Click de la clase IUSingular.

Como se aprecia en esta figura 6.9, el método: verifica el dato de entrada (líneas 09 a 16), crea el subproceso en .NET (líneas 28 y 29), registra (crea) el subproceso en el GASP (línea 29), registra los manejadores de eventos (líneas 32 a 34 y 49) y arranca el subproceso.

6.3 CASOS DEMOSTRATIVOS DE SPGs EN EL GASP.

De los diferentes escenarios que se pueden construir y mostrar con la aplicación SisGASP empleables en el proceso de enseñanza – aprendizaje de subprocesos y procesos de sistemas operativos, se presentan tres grupos relacionados con subprocesos. Después de ingresar al SisGASP, a través de su componente ejecutable IGASP, para cada grupo se destacan la configuración requerida del GASP, la preparación de la interfaz del SPG y las instancias significativas del avance de ejecución del SPG(s) por escenario. La figura 6.10 muestra formularios de configuración parcial del GASP y el formulario para la preparación de la interfaz del SPG,

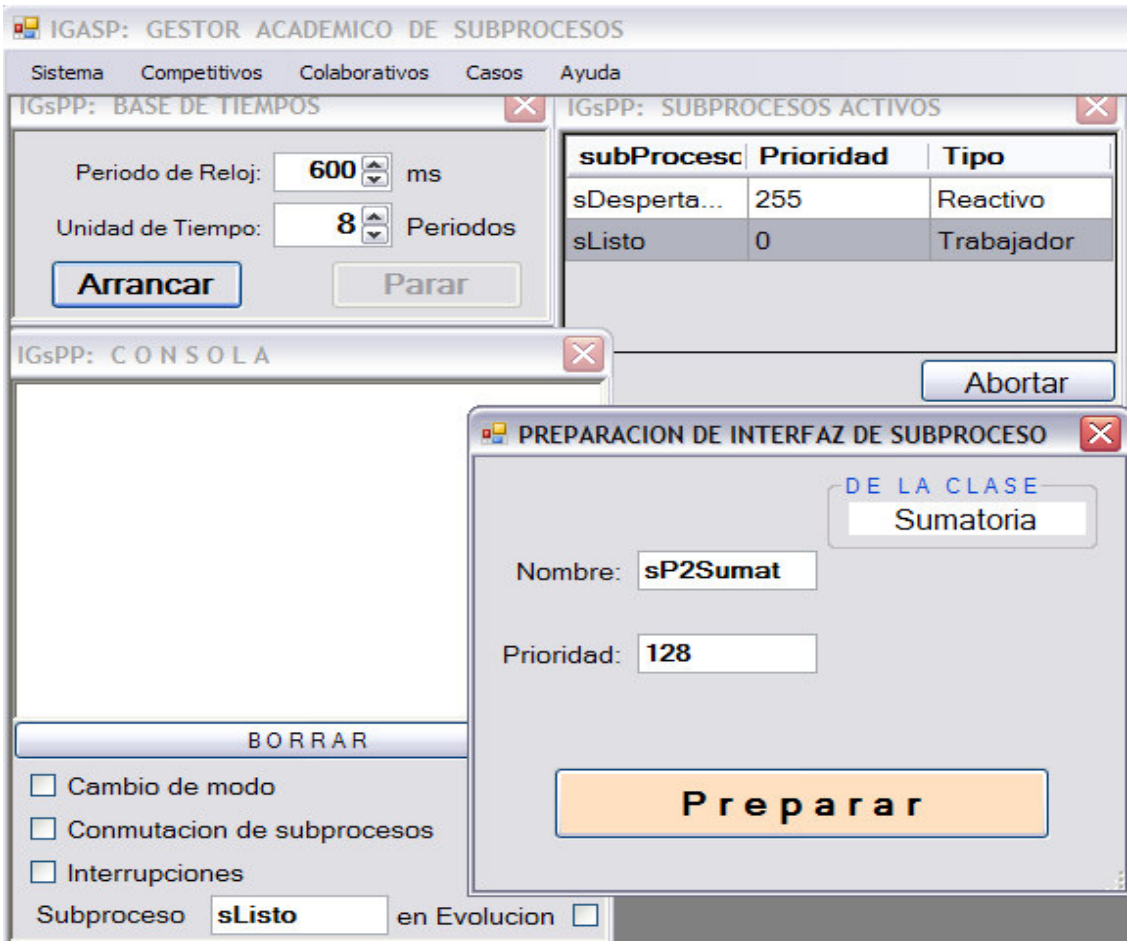


Figura 6.10 Configuración parcial de GASP y preparación de interfaz de SPG.

Por medio de la base de tiempos se arranca, se suspende y se reanuda el funcionamiento del GASP, que constituye una forma de suspender y reanudar la ejecución de los subprocesos gestionados. La consola permite ver cuatro características y sus combinaciones del funcionamiento del GASP gestionando subprocesos. Los

subprocesos gestionados activos se muestran en el formulario subprocesos activos. El formulario de preparación de interfaz muestra la clase del SPG, así como la memoria o búfer que usa, con su respectivo valor actual. Este último formulario, también, es el medio para ingresar el nombre y la prioridad del SPG y permite elegir los buzones requeridos por el subproceso. A los tres primeros formularios se ingresa por las opciones del menú Sistema, al último – por las opciones de los otros menús.

6.3.1 SUBPROCESOS INDEPENDIENTES.

Los subprocesos independientes compiten por los recursos del computador, dentro del proceso PGA, sin colaborar entre ellos. Dentro de este grupo, la aplicación incluye dos tipos de subprocesos gestionados: sumatoria y factorial, organizados en el menú Competitivos. Para los casos demostrativos siguientes, se emplea sumatoria.

Caso: un subproceso independiente. Creación, arranque, evolución de estado, ejecución, suspensión, ejecución y finalización de un subproceso Sumatoria.

a. En preparación de interfaz de subproceso, de la figura 6.10, se ajusta el nombre a sP2Sumat y se acepta la prioridad 128. Pulsando Preparar se crea la interfaz del subproceso sumatoria, que aparece en la figura 6.11.

b. La consola se prepara para ver los estados por los que pasa sP2Sumat durante su vida, indicando su nombre y marcando la respectiva casilla.

c. En la interfaz de sumatoria de la figura 6.11: se ingresa el número 5 y se pulsa Crear subproceso. Como consecuencia se crea sP2Sumat, se adiciona a subprocesos activos y se acredita en consola.

d. Pulsando el Arrancar de la base de tiempos en la figura 6.11, se inicia la ejecución de sP2Sumat. La figura 6.12 presenta el subproceso sP2Sumat en ejecución, mostrando: sus flujos de instrucciones y datos en su interfaz, sus estados en consola, su nombre en subprocesos activos y Parar de la base de tiempos se habilita. Parar de la base de tiempos permite suspender los subprocesos en ejecución.

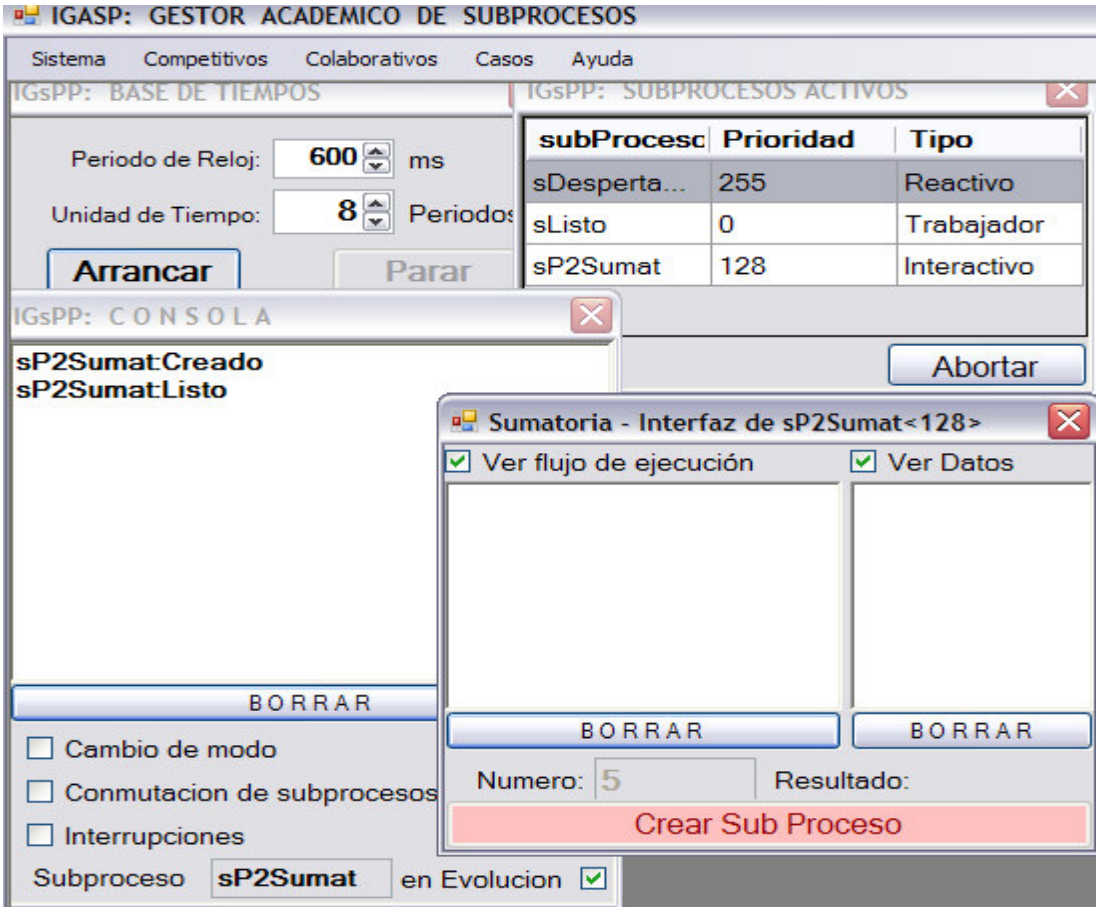


Figura 6.11 Subproceso sP2Sumat recién creado en SPG parado.

e. Mientras se ejecuta sP2Sumat, puede ser suspendido pulsando Parar de la base de tiempos. De forma análoga cuando sP2Sumat está suspendido, su ejecución puede reanudarse pulsando Arrancar de la base de tiempos. Esta dinámica se acredita en la interfaz de sP2Sumat, la consola y la base de tiempos.

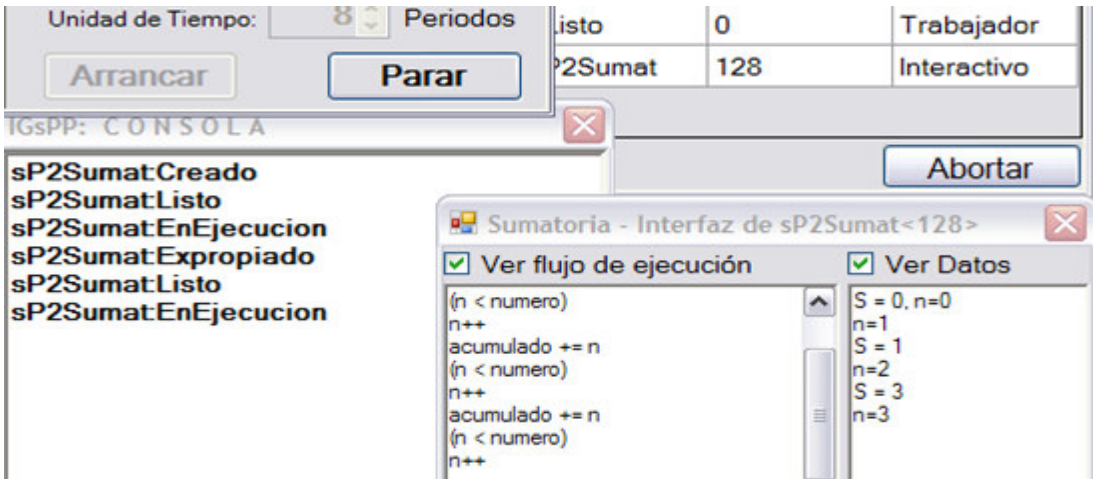


Figura 6.12 Subproceso sP2Sumat en ejecución.

f. La figura 6.13 acredita la finalización del subproceso: sP2Sumat desaparece de subprocesos activos con la respectiva indicación, sP2Sumat finalizado se muestra en consola y la interfaz de sumatoria presenta información final y habilita su comando para crear otro subproceso. El GASP continúa funcionando como lo muestra la interfaz de la base de tiempos.

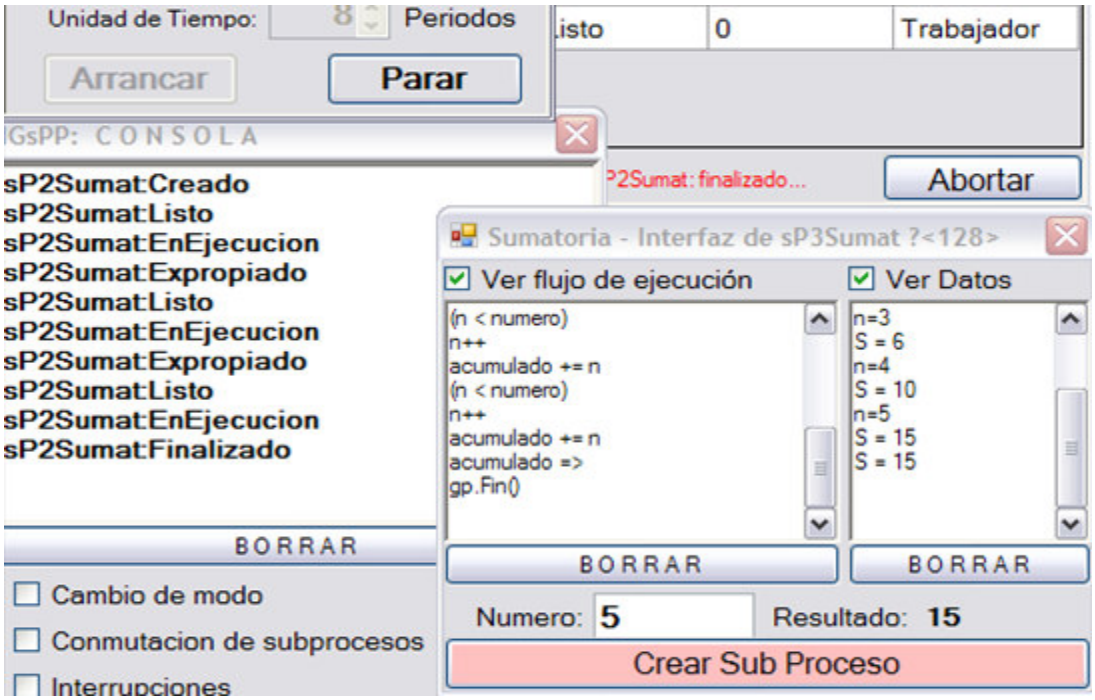


Figura 6.13 Subproceso sP2Sumat recién finalizado.

g. Este escenario puede repetirse con otras instancias de Sumatoria o de Factorial incorporados en la aplicación SisGASP.

Caso: múltiples subprocesos independientes con diferente prioridad. Creación, ejecución concurrente y finalización de dos subprocesos del tipo sumatoria. Se puede reproducir el mismo escenario anterior para este caso; sin embargo, para simplificar contenidos, ya no se visualizan la base de tiempos y los subprocesos activos. La figura 6.14 presenta el resumen gráfico del siguiente escenario:

a. Iniciar a SisGASP: Correr IGASP → menú Sistema → Base de Tiempos → pulsar Arrancar → Cerrar formulario Base de Tiempos.

b. Crear interfaz de subproceso sumatoria con menor prioridad: menú Competitivos → Sumatoria → pulsar Preparar, aceptando nombre y prioridad propuestos.

c. Crear interfaz de subproceso sumatoria con mayor prioridad: menú Competitivos → Sumatoria → aceptar nombre propuesto e ingresar 130 en prioridad → pulsar Preparar.

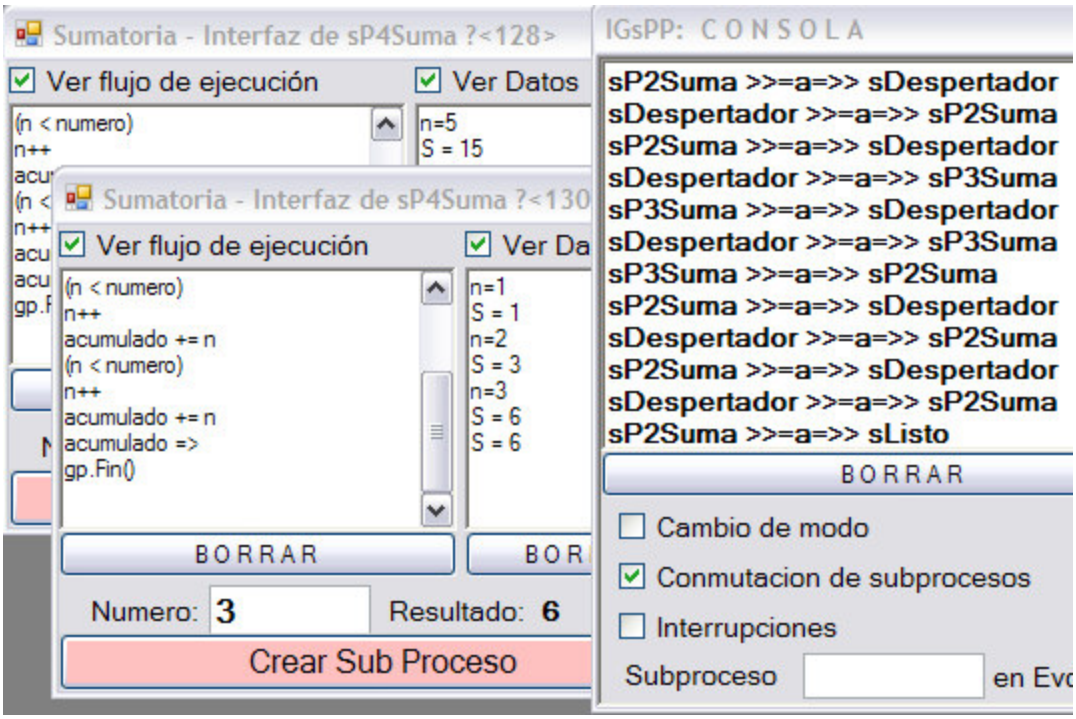


Figura 6.14 Ejecución de dos subprocesos con prioridades diferentes en GASP

d. Abrir consola: menú Sistema → Consola → marcar la casilla Conmutación de subprocesos.

e. Crear subproceso con menor prioridad, ingresando número 7 y pulsando Crear subproceso en primera interfaz. Se crea sP2Suma y empieza a ejecutarse tal como se muestra en su interfaz y consola.

f. Crear subproceso de mayor prioridad, ingresando número 3 y pulsando Crear subproceso en segunda interfaz. Se crea sP3Suma y se ejecuta hasta finalizar, suspendiendo la ejecución de sP2Suma. Al finalizar sP3Suma, se reanuda sP2Suma y se ejecuta hasta finalizar. Esta dinámica se aprecia en ambas interfaces y en la consola. El GASP primero sirve a los subprocesos de mayor prioridad, los procesos de la misma prioridad son servidos por orden de llegada (creación).

g. Se pueden crear mas subprocesos con la misma o diferente prioridad empleando cualquier opción del menú Independientes.

6.3.2 SUBPROCESOS COLABORATIVOS.

La aplicación SisGASP implementa múltiples subprocesos gestionados que adicionan (acumulan) números a una dirección de memoria en forma colaborativa, incorporados en el menú colaborativos en dos grupos: del mismo tipo (clase) y de tipos complementados. Cada subproceso del mismo tipo recibe un número del usuario y lo acumulan en la misma dirección de memoria. Las dos primeras opciones del mismo tipo crean acumuladores no sincronizados y la última crea acumuladores sincronizados por exclusión mutua usando mensajes. En los subprocesos complementados, uno crea mensajes de sincronización y los números que el otro recibe vía una dirección de memoria compartida o como parte del mismo mensaje. El primer caso es una señalización por mensaje y el segundo – es un mensaje de comunicación de datos. En ambos casos se requieren dos buzones. De los tantos escenarios que se pueden construir se presentan los siguientes casos:

Caso: compartición no sincronizada de memoria. Dos acumuladores no sincronizados actualizan una memoria compartida: uno con 3 y otro con 5, esperando obtener un acumulado de 8. Si el resultado no es 8, se tiene un problema de sincronización. La figura 6.15 muestra el resumen gráfico de este escenario:

a. Ingresar a SisGASP.

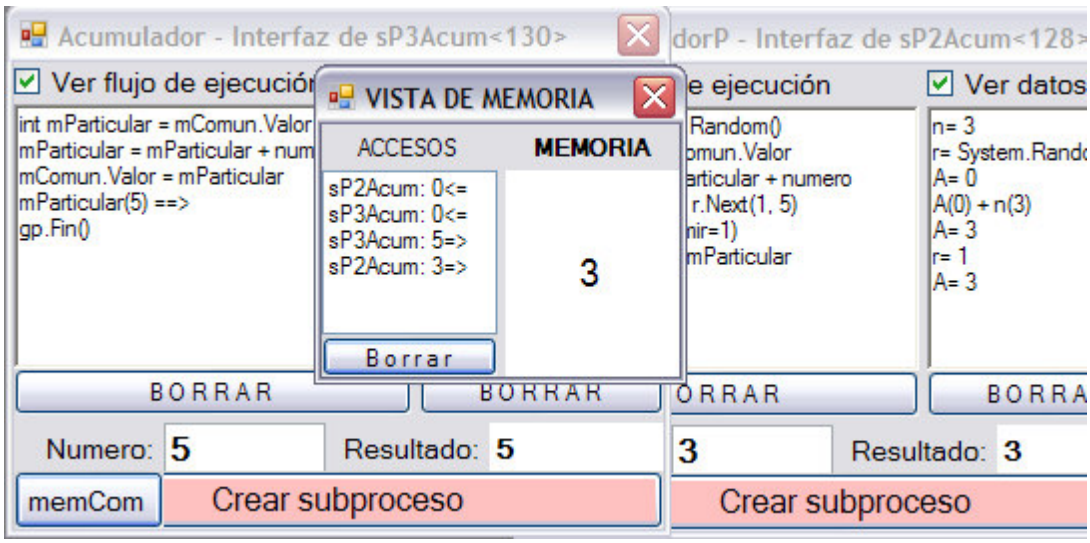


Figura 6.15 Compartición de memoria por dos subprocesos no sincronizados en GASP

b. Crear interfaz de acumulador con mayor prioridad: menú Colaborativos → Un Tipo → Acumulador → aceptar nombre propuesto e ingresar 130 en prioridad → pulsar Preparar. Se muestra también la memoria compartida con valor 0.

c. Crear interfaz de Acumulador extendido con menor prioridad: menú Colaborativos → Un Tipo → acumulador Ext.→ aceptar nombre y prioridad propuestos → pulsar Preparar. Borrar accesos en vista de memoria compartida.

d. Crear acumulador de menor prioridad, ingresando 3 en número y pulsando Crear subproceso. sP2Acum se crea y empieza a ejecutarse como se ve en su interfaz. Inmediatamente ejecutar siguiente paso.

e. Crear acumulador de mayor prioridad, ingresando 5 en número y pulsando Crear subproceso. sP3Acum se crea y se ejecuta hasta finalizar, suspendiendo a sP2Acum, como se aprecia las interfaces.

f. La ejecución de sP2Acum se reanuda y finaliza. El resultado final en la memoria compartida es 3, en lugar de 8, error que se resuelve con sincronización.

Caso: compartición sincronizada de memoria por exclusión mutua. Dos acumuladores sincronizados actualizan una memoria compartida: uno con 3 y otro con 5. Independiente del orden en que se creen y de las prioridades que tengan el resultado será 8. La figura 6.16 muestra el resumen gráfico de este escenario:

a. Ingresar a SisGASP.

b. Adicionar un buzón para mensaje de sincronización: menú Sistema → Buzones → pulsar Adicionar. Se adiciona el buzón 8 a la lista de buzones. Cerrar la vista de buzones.

c. Adicionar el mensaje de sincronización: menú Sistema → Mensajes → seleccionar buzón 8 → pulsar Adicionar. El mensaje se adiciona al buzón 6. Cerrar la vista de manejo de mensajes.

d. Crear interfaz de primer acumulador sincronizado: menú Colaborativos → Un Tipo → Acumulador Sinc → aceptar nombre y prioridad propuestos y marcar buzón 8 → pulsar Preparar. Se muestra: la interfaz del acumulador, la memoria compartida con valor 0 y la vista del buzón 8 con el mensaje de bloqueo de esta memoria compartida.

e. Crear interfaz de segundo acumulador sincronizado: menú Colaborativos → Un Tipo → Acumulador Sinc → aceptar nombre y prioridad propuestos y marcar buzón 8 → pulsar Preparar. Se muestra la interfaz de este segundo acumulador.

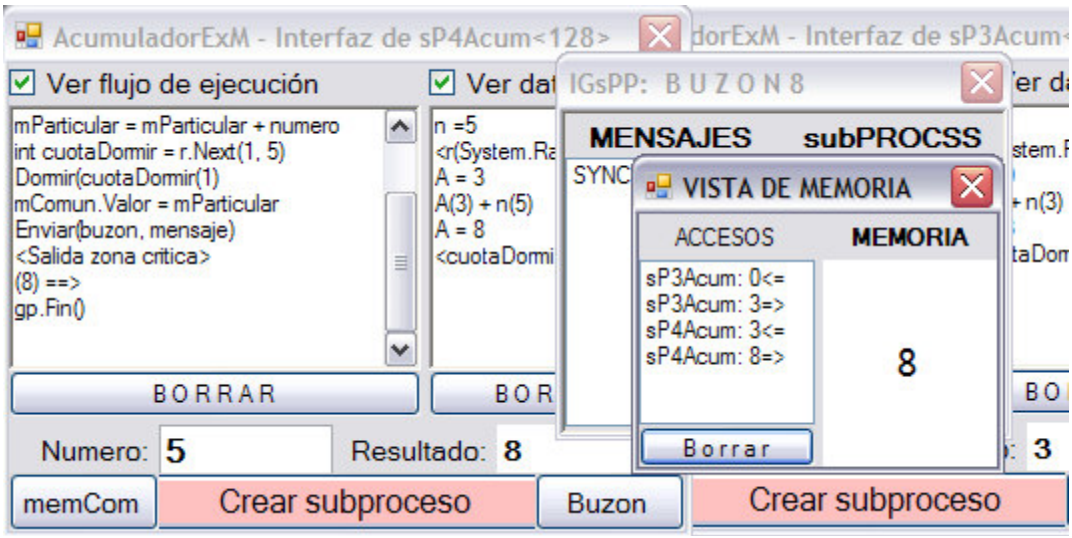


Figura 6.16 Compartición de memoria por dos subprocesos sincronizados en GASP

f. Crear primer acumulador, ingresando 3 en número y pulsando Crear subproceso. sP3Acum se crea y empieza a ejecutarse, retirando el mensaje del buzón, como se ve en las interfaces. Inmediatamente ejecutar siguiente paso.

g. Crear segundo acumulador, ingresando 5 en número y pulsando Crear subproceso. sP4Acum se crea y al ejecutarse busca el mensaje en el buzón. Si lo encuentra continúa, sino espera hasta que sP3Acum finalice y regrese el mensaje al buzón..

h. El resultado final en la memoria compartida será el correcto, para este caso es 8.

6.3.3 SUBPROCESOS PRODUCTORES Y CONSUMIDORES.

Dentro del menú Casos de la aplicación SisGASP se encuentra el submenú Productores/Consumidores. La primera opción de este submenú conduce a crear productores y consumidores que colaboran incorrectamente, las otras tres – conducen a crear productores y consumidores sincronizados que colaboran correctamente. De los diferentes escenarios que se pueden elaborar, se presentan los siguientes:

Caso: Productor y consumidor con un búfer de uso mutuamente exclusivo. Se emplea un mensaje de sincronización para indicar la disponibilidad del búfer y dos buzones. La presencia del mensaje en el primer buzón hace disponible el búfer para el consumidor y en el segundo – para el productor. Para iniciar el proceso de producción y consumo, el usuario debe ingresar el mensaje de sincronización en el segundo buzón. La figura 6.17 muestra el resumen gráfico del escenario.

a. Ingresar a SisGASP.

b. Adicionar dos buzones: menú Sistema → Buzones → pulsar Adicionar dos veces. Se agregan dos buzones: el 8 (primer buzón) y el 9 (segundo buzón). Cerrar la vista con lista de buzones.

c. Adicionar mensaje de sincronización en buzón 9: menú Sistema → Mensajes → seleccionar buzón 9 → pulsar Adicionar. Mensaje se ingresa en el buzón 9. Cerrar la vista de manejo de mensajes.

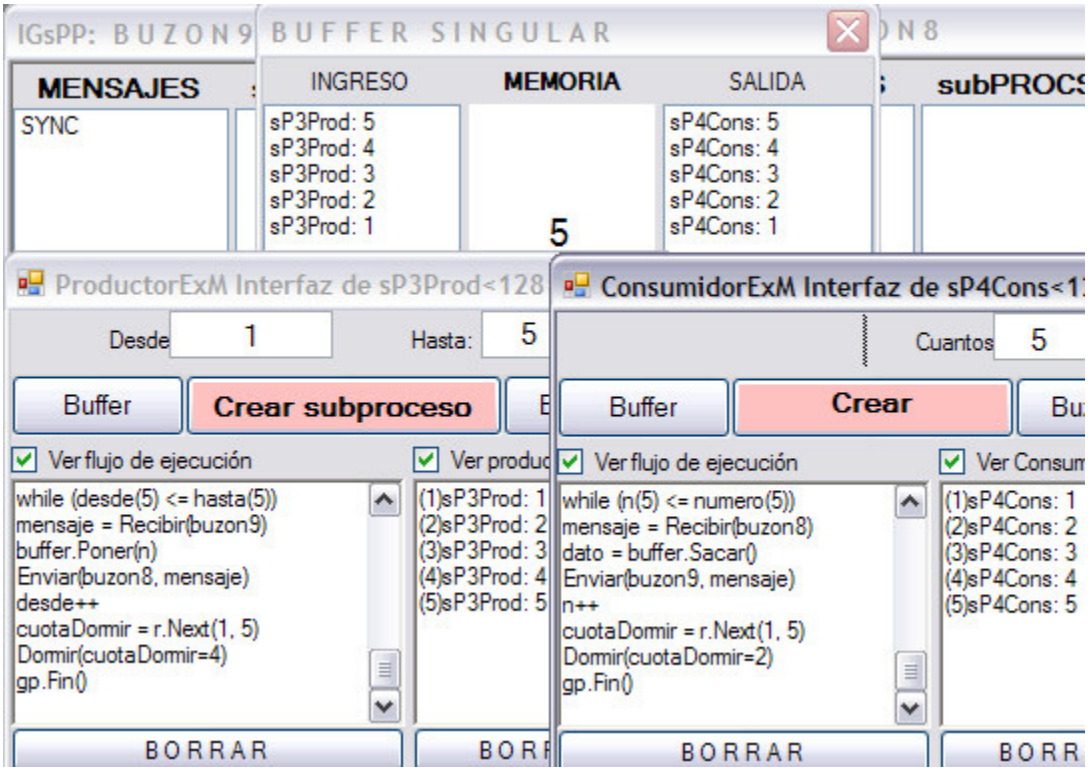


Figura 6.17 Producción y consumo sincronizados vía un búfer c/exc.mutua en GASP

d. Crear interfaz de productor: menú Casos → Produc/Consum → Exclusión Mutua → Productor →aceptar nombre y prioridad propuestos y marcar buzones 8 y 9 → pulsar Preparar. Se muestran: la interfaz del productor, el búfer y el buzón 9 con el mensaje de sincronización.

e. Crear interfaz de consumidor: menú Casos → Produc/Consum → Exclusión Mutua → Consumidor →aceptar nombre y prioridad propuestos y marcar buzones 8 y 9 → pulsar Preparar. Se adicionan: la interfaz del consumidor y el buzón 8.

f. Crear productor: ingresar 1 en desde y 5 en hasta, luego pulsar Crear subproceso. Se crea el subproceso sP3Prod, produce los números 1, 2, 3, 4, 5, y finaliza.

g. Crear consumidor: ingresar 5 en cuantos y pulsar Crear subproceso. Se crea el subproceso sP4Cons que consume correctamente los números producidos por sP3Prod y finaliza.

Caso: Productores y consumidores con doble canal (buzón) y búfer ilimitado de uso mutuamente exclusivo. El productor genera aleatoriamente un número con el cual arma un mensaje que es remitido al segundo buzón, el número también es colocado en el búfer ilimitado. El productor primero retira el mensaje del segundo buzón y luego del búfer ilimitado. Para ambos subprocesos, la presencia de un mensaje de sincronización en el primer buzón indica la disponibilidad del búfer. El mensaje de sincronización es insertado en el primer buzón por el usuario. La figura 6.18 resume gráficamente este escenario.

a. Ingresar a SisGASP.

b. Adicionar dos buzones: menú Sistema → Buzones → pulsar Adicionar dos veces. Se agregan dos buzones: el 8 (primer buzón) y el 9 (segundo buzón). Cerrar la vista con lista de buzones.

c. Adicionar mensaje de sincronización en buzón 8: menú Sistema → Mensajes → seleccionar buzón 8 → pulsar Adicionar. Mensaje se ingresa en el buzón 8. Cerrar la vista de manejo de mensajes.

d. Crear interfaz de productor: menú Casos → Produc/Consum → Comunicación Datos → Productor →aceptar nombre y prioridad propuestos y marcar buzones 8 y 9 → pulsar Preparar. Se muestran: la interfaz del productor, el búfer, el buzón 8 con el mensaje de sincronización y el buzón 9 para trasmisión de datos.

e. Crear interfaz de consumidor: menú Casos → Produc/Consum → Comunicación Datos → Consumidor →aceptar nombre y prioridad propuestos y marcar buzones 8 y 9 → pulsar Preparar. Se adiciona la interfaz del consumidor.

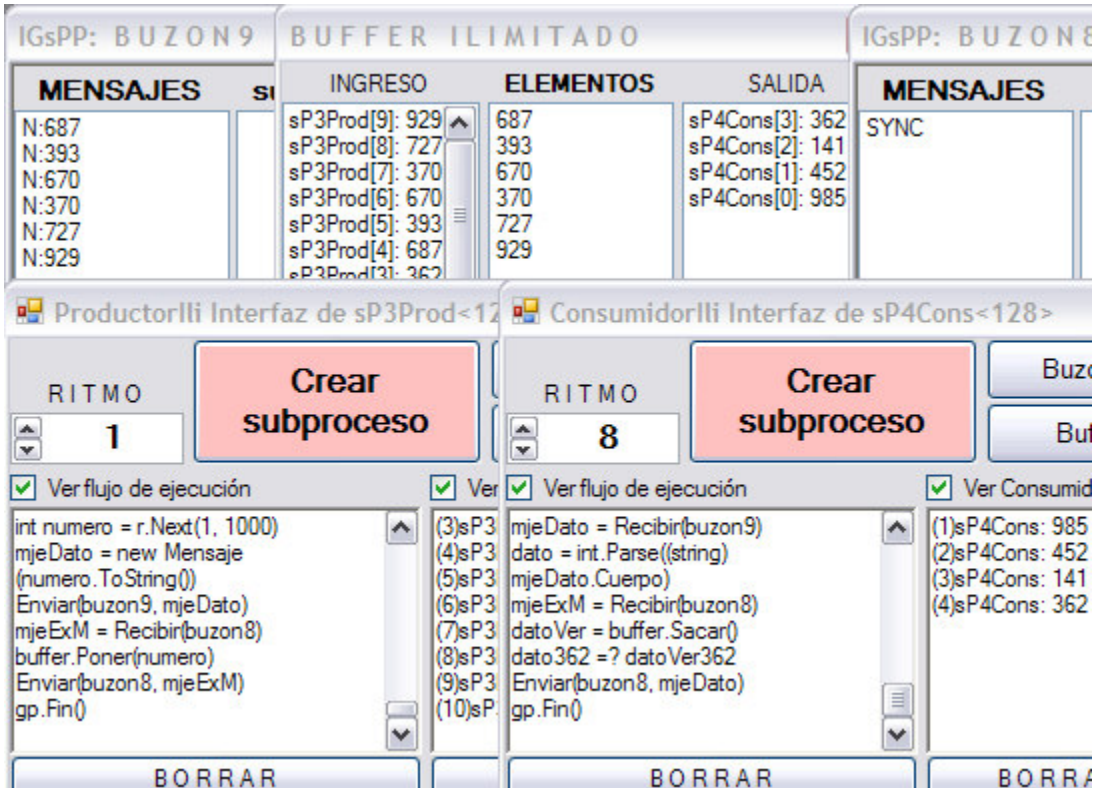


Figura 6.18 Producción y consumo sincronizados vía buzón y búfer ilimitado en GASP

h. Crear productor: ingresar 1 en ritmo y pulsar Crear subproceso. Se crea sP3Prod y produce números aleatorios que lo coloca en el buzón 9 y el búfer .

f. Crear consumidor: ingresar 8 en ritmo, para que se ejecute a menor velocidad que el productor, y pulsar Crear subproceso. Se crea el subproceso sP4Cons que consume correctamente los números producidos por sP3Prod.

g. Dada la diferencia de las velocidades de ejecución, el productor se ejecuta más rápido que el consumidor, la relación de números en el búfer y la cola de mensajes en el buzón 9 empiezan a crecer. Esta situación puede cambiarse, modificando los ritmos de ejecución de los subprocesos.

h. Después de un momento se pulsan los botones Terminar subproceso de productor y consumidor y los subprocesos finalizan.

En el siguiente capítulo se especifican las principales conclusiones y sugerencias, relacionadas con este trabajo.

Capitulo VII:
CONCLUSIONES Y SUGERENCIAS.

Al finalizar el presente trabajo, realizado en el marco presentado en los tres primeros capítulos y expresado en los últimos tres, se formulan las conclusiones sobre las características del gestor académico de subprocesos (GASP), su desarrollo seguido y su aplicación; así mismo se plantean algunas posibilidades en la forma de sugerencias.

7.1 CONCLUSIONES.

Para el empleo en el proceso de enseñanza – aprendizaje de subprocesos y procesos en los sistemas operativos, **las principales características del GASP**, especificadas en el capítulo 4, incluyen:

- Planificación básica de subprocesos, por medio de las operaciones: PLANIFICAR SUBPROCESO y DESPACHAR SUBPROCESO.
- Sincronización y comunicación de subprocesos, por medio de las operaciones: ENVIAR MENSAJE, RECIBIR MENSAJE, INSERTAR BUZÓN, ELIMINAR BUZÓN y LEER BUZONES.
- Gestión de interrupciones, por medio de las operaciones HABILITAR NIVEL DE INTERRUPCION, SERVIR INTERRUPCIONES. La operación DESPACHAR SUBPROCESO participa estableciendo la máscara de interrupciones del CIP para el subproceso que se despacha.
- Administración de subprocesos, por medio de las operaciones: CREAR SUBPROCESO, FINALIZAR SUBPROCESO, ABORTAR SUBPROCESO, DORMIR SUBPROCESO, DESPERTAR SUBPROCESO. Según sus estados básicos, los subprocesos se organizan en listas: todos, listos y dormidos. La lista de todos los subprocesos se administra por las operaciones: insertar, eliminar, buscar y leer. Las listas de listos y dormidos se administran por las operaciones insertar y eliminar.
- Configuración inicial, por medio de la operación CONFIGURACION INICIAL.
- Ejecución referencial de los subprocesos al lenguaje de programación general, orientado a objetos, C#.
- Temporización (base de tiempos) y control de la velocidad de ejecución de los subprocesos referenciados al lenguaje C#, en el orden de segundos.

- Control de la dinámica de ejecución de los subprocesos, por medio de la administración de buzones y mensajes.
- Interfaz gráfica del GASP, para manejar: su base de tiempos, sus buzones, sus mensajes y visualizar su dinámica, que es la dinámica de los subprocesos gestionados.
- Interfaz gráfica de subproceso, para administrar el subproceso y controlar la visualización del avance de su ejecución.
- Instalación en la plataforma común .NET 1.1 y compatibles, incorporada en los sistemas operativos vigentes de Microsoft, entre los que se encuentran: Windows xp, Windows Vista, Windows 7.

El proceso de desarrollo del GASP ha seguido un enfoque orientado a objetos, destacándose las disciplinas de requisitos, diseño, implementación y pruebas.

Los requisitos funcionales y no funcionales han sido especificados en el capítulo IV, en la forma de características del GASP, a partir del objetivo general y objetivos específicos planteados en la tesis. El capítulo V describe el diseño e implementación del GASP y de la interfaz de integración, y el capítulo VI - la construcción de los subprocesos y las pruebas.

El diseño ha sido modelado a nivel arquitectónico y a nivel de clases y objetos, empleando UML en la herramienta Microsoft Office Visio 2007, particularmente diagramas estructurales. El GASP se ha ubicado dentro del SisGASP, una aplicación desktop de mayor alcance que incluye los subprocesos gestionados y la interfaz (IGASP) que integra estos subprocesos con el GASP.

El diseño arquitectónico concibe el GASP, cada subproceso gestionado y la interfaz IGASP en capas, constituidas por paquetes de clases y objetos:
- El GASP comprende dos capas: presentación (IGSPP) y lógica de aplicación (GSPP). Las clases y objetos del paquete IGestion constituyen la capa IGSPP. GSPP está compuesta por seis paquetes de clases y objetos: Mascara de Interrupciones, elementos básicos (EEBB), Listas de elementos básicos, Emulador de Hardware, Núcleo y Tareas Específicas..
- Cada subproceso gestionado interactivo comprende dos capas: presentación (ISP) y lógica de aplicación (SP). Los subprocesos no interactivos no requieren la capa ISP. La naturaleza de la tarea de cálculo define la estructura de cada capa del subproceso, el cual accede al GASP por medio de la clase Tarea.

- La IGASP comprende una capa: presentación, que contiene las clases IGI y IUpreIUsp. Este subsistema IGASP facilita al usuario final el acceso a las interfaces del GASP y de los subprocesos, e integra al GASP y los subprocesos en la aplicación desktop SisGASP, ejecutable sobre .NET.

Consistente con su diseño, el GASP ha sido implementado en un proyecto de programación en el IDE Microsoft vs.Net 2005 con lenguaje C#, que es uno de los tres proyectos construidos en la solución SisGASP, correspondientes a los tres subsistemas de implementación que constituyen la aplicación desktop SisGASP:

- GASP, con los componentes de producción correspondientes a las clases diseño del GASP, implementado en un proyecto del tipo librería de clases que construye el componente desplegable GASP.DLL.
- IGASP, con los componentes de producción que se trazan a las clases de diseño de la IGASP, implementado en un proyecto del tipo aplicación Windows que construye el componente desplegable IGASP.EXE.
- subProcesos, organiza los componentes de producción que implementan las clases de diseño de los subprocesos en un proyecto del tipo librería de clases que construye el componente desplegable subProcesos.DLL

Las pruebas del GASP, de la IGASP y de subProcesos han sido realizadas a nivel unitario, a nivel de integración y a nivel de sistema. Los procedimientos de algunas de las pruebas de sistema se especifican en casos demostrativos de subprocesos gestionados en el GASP en el capítulo anterior.

La **aplicación del GASP** ha sido materializada dentro del programa desktop SisGASP como una plataforma sobre la cual se han construido un conjunto de subprocesos. SisGASP facilita la presentación visual de las características de los subprocesos gestionados y del proceso (PGA) que los contiene, a voluntad del usuario, para ser empleados en el proceso de enseñanza – aprendizaje de subprocesos y procesos de sistemas operativos.

Los siguientes aspectos del proceso de ejecución del SisGASP (PGA) son mostrables: base de tiempos, buzones, mensajes, subprocesos activos y consola con dinámica de ejecución de subprocesos.

Para cada subproceso se crea un interfaz, a partir de la cual se crea el subproceso y se puede abortarlo, y en la cual es se visualiza el avance de ejecución referenciada del subproceso en términos de instrucciones en lenguaje C# y de resultados parciales.

El conjunto de subprocesos construidos contiene subprocesos independientes, subprocesos colaborativos que comparten una dirección de memoria y subprocesos productores y consumidores.

Entre los subprocesos independientes se han incluido la sumatoria y factorial.

Como subprocesos colaborativos que comparten memoria se han involucrado diferentes versiones de acumuladores de números en una dirección de memoria.

Los subprocesos productores y consumidores extienden las actividades de colaboración, presentes en los sistemas operativos.

Los escenarios, caracterizados por varios procesos con sus respectivos subprocesotes, se alcanzan creando múltiples instancias del SisGASP.

7.2 SUGERENCIAS.

El empleo significativo del GASP requiere conocimientos básicos sobre la gestión de procesos y subprocesos de los sistemas operativos; así como sobre los factores que participan en el proceso de enseñanza – aprendizaje, particularmente de la velocidad de exposición del material que se usa en este proceso. Después de conocer el GASP y experimentar todo lo que facilita la aplicación desktop SisGASP, se derivan algunas posibilidades que se sugieren explorar:

Construir nuevos subprocesos para otras tareas de cómputo y adicionarlos a SisGASP dentro de los grupos de tipos establecidos: independientes, colaborativos o casos muy tipificados.

Implementar el GASP y el SisGASP sobre otra plataforma con otro lenguaje de programación como JVM con lenguaje Java.

Variar algunos requisitos como el reloj. Incorporar múltiples relojes y asociarlos con familias de subprocesos o subprocesos individuales.

Adicionar características gráficas y de audio a las interfaces de los subprocesos en la aplicación desktop SisGASP para mostrar los resultados intermedios en multimedia.

Extender el alcance de la aplicación del GASP más allá de procesos y subprocesos de sistemas operativos, a otras áreas tales como: programación, lógica, matemáticas, pedagogía, didáctica.

Extender el GASP para cubrir otros aspectos de gestión de procesos y subprocesos tales como: gestión de procesos, respaldo y recuperación de contexto, fuentes de interrupción diversas, diferentes estrategias de planificación de procesos y subprocesos.

LISTA DE CUADROS

2.1 Ordenes típicas de un sistema operativo por clase funcional.

2.2 Capas del sistema operativo.

LISTA DE FIGURAS

1.1 Modelo de condicionamiento.

1.2 Modelo de condicionamiento operante.

1.3 Modelo de aprendizaje E-O-R.

1.4 Modelo de teorías de procesamiento de información de Gagné.

1.3 Modelo de aprendizaje E-M-O-M-R.

1.6 Modelo del proceso de enseñanza – aprendizaje de Ferrández.

2.1 Computador o sistema informático.

2.2 Arquitectura del computador “modelo de Von Newman”.

2.3 Arquitectura del computador con buses.

2.4 Procesador (µP).

2.5 Memoria primaria RAM.

2.6 Computador con periféricos.

2.7 Estructura genérica de la interfaz.

2.8 Puerto genérico de E/S.

2.9 Entrada controlada por programa.

2.10 Señales de interrupciones.

2.11 SO, programas y sus interfaces.

2.12 Multiprogramación.

2.13 Esquemas de procesos.

2.14 Gestión de procesos, hilos y recursos.

2.15 Gestión de memoria.

2.16 Gestión de dispositivos.

2.17 Gestión de archivos.

2.18 Requisitos funcionales de SO.

3.1 Máquinas real y abstractas.

3.2 Interfaz de la máquina abstracta.

3.3 Estados de proceso/hilo.

3.4 Gestor típico de recursos.

3.5 Planificación de procesos/hilos.

3.6 Planificador.

3.7 Cambios de contexto.
3.8 Comportamiento del temporizador programable.
3.9 Cruce de vías.
3.10 Secciones críticas.
3.11 Implementación de P y V con espera activa.
3.12 Sincronización con semáforos.
3.13 Implementación de semáforos con colas.
3.14 Sincronización simultánea.
3.15 Monitor.
3.16 Monitor implementado con semáforos.
3.17 Formato de mensaje.
3.18 Denominación en mensajes.
3.19 Intercambio de mensajes con espera temporizada.
3.20 Sincronización y comunicación con mensajes.
3.21 Interbloqueo entre dos procesos/hilos.
4.1 Transiciones de estado de subprocesos.
4.2 Relación entre distancia y comprensión.
4.3 Velocidad de ejecución de programas y comprensión de su dinámica.
4.4 Interfaz gráfica de usuario final.
5.1 Contexto del gestor académico de subprocesos.
5.2 Arquitectura por capas del GASP en su contexto.
5.3 Arquitectura en paquetes del gestor académico de subprocesos.
5.4 Clasificadores de máscara de interrupciones.
5.5 Dispositivos del computador emulados por el GASP.
5.6 Clasificadores de elementos básicos del GASP.
5.7 Clasificadores que manejan listas de elementos básicos.
5.8 Diagrama de clases del GASP.
5.9 Interfaz de usuario de la base de tiempos.
5.10 Interfaz de usuario de buzón.
5.11 Interfaz de gestión de lista de buzones.
5.12 Interfaz de administración de mensajes por buzón.
5.13 Consola de GASP.
5.14 Monitor de subprocesos.
5.15 Interfaz de memoria compartida.

5.16 Interfaz de búfer de una memoria.
5.17 Interfaz de búfer de un número limitado de memorias.
5.18 Interfaz de búfer sin limitación de memorias.
5.19 Interfaz de integración del GASP: IGASP.
5.20 Clase de servicio de interfaz IGASP.
5.21 Interfaz de preparación de parámetros de subproceso gestionado interactivo.
5.22 Arquitectura contextual de la implementación del GASP.
5.23 Paquetes de componentes de producción del GASP.
5.24 Componentes en máscara de interrupción.
5.25 Componentes en hard.
5.26 Componentes en EEBB.
5.27 Componentes en listas.
5.28 Componentes en núcleo.
5.29 Componentes en IGestor.
5.30 Componentes en tareas específicas.
5.31 Componentes del subsistema IGASP.
6.1 Arquitectura en capas del subproceso gestionado en su contexto.
6.2 Interfaz del usuario final del subproceso gestionado.
6.3 Lógica de aplicación de subproceso gestionado.
6.4 Estructura del SPG “sumatoria”;
6.5 Paquetes del subsistema “subProcesos” y componentes del SPG “sumatoria”.
6.6 Código fuente de la clase “Tarea” del subsistema GASP.
6.7 Clase “Sumatoria” en programa convencional.
6.8 Código fuente de la clase “Sumatoria” del SPG “sumatoria”.
6.9 Extracto del código fuente del método “bnCrear_Click” de la clase “IUSingular”.
6.10 Configuración parcial del GASP y preparación de interfaz de SPG.
6.11 Subproceso “sP2Sumat” recién creado en SPG parado.
6.12 Subproceso “sP2Sumat” en ejecución.
6.13 Subproceso “sP2Sumat” recién finalizado.
6.14 Ejecución de dos subprocesos con prioridades diferentes en GASP.
6.15 Compartición de memoria por dos subprocesos no sincronizados en GASP.
6.16 Compartición de memoria por dos subprocesos sincronizados en GASP.
6.17 Producción y consumo sincronizados vía búfer con exclusión mutua en GASP.
6.18 Producción y consumo sincronizados vía buzón y búfer limitado en GASP

.

BIBLIOGRAFIA.

[1] ANICAMA J. La psicología y el diseño científico de sistemas de enseñanza. Universidad Ricardo Palma. Escuela de PostGrado. Maestría en Docencia Universitaria. Lima (Perú). [200?].

[2] AUSUBEL, D.P.; NOVAK, J.; HANESIAN H. Psicologia educativa. Un punto de vista cognoscitivo. Trillas, Mexico D.F., 1976.

[3] BOOCH G.; RUMBAUGH J.; JACOBSON I. El lenguaje unificado de modelado. 2ª ed. Madrid (España). 2006.

[4] BREY B.B. Los microprocesadores intel avanzados. Limusa, Mexico (DF), 1994.

[5] DEITEL H.M. Introducción a los sistemas operativos. 2a ed. Addison Wesley, Delaware (EUA), 1993.

[6] DEITEL H.M.; et al. C# how to program. Prentice Hall, New Jersey (USA), 2002.

[7] Fundamentos psicológicos del proceso de enseñanza aprendizaje. [s.l.]:[s.n.]. [199?]. [En línea]. [Consulta -03-2010]. Formato PDF. Disponible en web: http://www.docstoc.com/docs/2778104/Fundamentos_Psicopedagogicos.

[8] JACOBSON I.; BOOCH G.; RUMBAUGH J. El proceso unificado de desarrollo de sotware. Addison Wesley, Madrid (España), 2000.

[9] LARMAN C. APPLYING UML AND PATTERNS. 3rd ed. Prentice Hall, New Jersey (USA), 2005.

[10] MANO M.M. Computer system architecture. 3rd ed. Prentice Hall, New Jersey (USA), 1993.

[11] MENESES B.G. NTIC, interacción y aprendizaje en la Universidad. [en línea]. [España]:Universidad Rovira I Virgili. 2008. [consulta: -04-2010]. Documento: ElProcesoDeEnseñanza. Tesis: defendica el 26-06-2007. Formato pdf. Disponible en web: http://www.tdx.cat/TDX-1207107-161635. DL/ISBN: T.2183-2007/978-84-691-0359-3.

[12] MILENKOVIC, M. Sistemas operativos: Conceptos y diseño. 2a ed. McGraw-Hill, Madrid (España), 1994.

[13] MORGAN C.L.; WAITE M. Introducción al microprocesador 8086/8088. McGraw-Hill. Naucalpan de Juarez (Mexico), 1984.

[14] NUTT, G. Sistemas Operativos. 3a ed. Pearson:Addison Wesley, Madrid (España), 2004.

[15] TANENBAUM, A.S. Structured computer organization. 4th ed. Prentice-Hall, New Jersey (USA), 1999.

[16] TANENBAUM, A.S. Sistemas operativos modernos. [en linea] 2a ed. Naucalpan de Juarez (Mexico): Prentice-Hall. 2003. [consulta: 2009]. Formato pdf. Vista previa restringida, disponible en web: http://books.google.com.pe/books?id=g88A4rxPH3wC&printsec=frontcover&source=gbs_v2_summary_r&cad=0#v=onepage&q&f=false. ISBN: 970-26-0315-3.

[17] TANENBAUM, A.S.; WOODHULL A.S. Sistemas operativos: Diseño e implementación. 2a ed. Prentice-Hall, Naucalpan de Juarez (Mexico), 1998.

[18] TRILLA, J.; et al. El legado pedagógico del siglo XX para la escuela del siglo XXI. [en línea]. Barcelona (España): Grao. 2001. [consulta: -02-2010]. Formato pdf. Vista previa restringida, disponible en web: http://books.google.com.pe/books?id=31urauk4NSgC&printsec=frontcover&source=gbs_v2_summary_r&cad=0#v=onepage&q&f=false. DL/ISBN: B-21.058-2007 / 978-84-7827-256-3.

[19] ZAVALZA, M.A. Las diez dimensiones de una docencia de calidad. En su: Competencias docentes del profesorado universiatrio. Calidad y desarrollo professional. 2a ed. Narcea, Madrid (España), 2003.

ANEXOS:

LISTADOS DE PROGRAMAS DEL sisGASP.

A. Fuentes de GASP en C#.

//UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
//FACULTAD DE CIENCIAS MATEMATICAS
//UNIDAD DE POST-GRADO
//MAESTRIA EN COMPUTACION E INFORMATICA
//TESIS DE GRADO: Ing. Francisco S. Aguilar V.
//FECHA: 2011
//Subsistema GASP – Gestor Académico de SubProcesos

///Paquete: Mascara Interrupción – máscara de interrupciones.

////Componente: MascaraInt1.cs

```
using System;
using System.Collections;
using System.Text;

namespace GASP
{
  /* La mascara de interrupciones del sistema esta compuesta
   * por ocho niveles de interrupcion, enumeradas del 0 al 7,
   * y está intimamente relacionada con los niveles de inter-
   * rupción hardware del Controlador de Interrupciones Progra-
   * mable.
   *
   * Las prioridades manejadas por el sistema están en el ran-
   * go de 0 a 255. La máxima prioridad es 255 y la mínima es
   * 0. Las n prioridades superiores corresponden al nivel
   * de interrupcion 0: n prioridades por nivel; n = 4
   *
   * Los sub/procesos de prioridad mayor expropian la CPU
   * a los sub/procesos de menor prioridad. Se disponen de las
   * 64 mayores prioridades para los sub/procesos de servicio
   * a las interrupciones hardware, manejadas por el CIP
   *
   * Para fines de mejor entendimiento usaremos logica positi-
   * va: 1 = habilitar interrupcion y 0 = deshabilitar.
   *
   * La siguiente tabla relaciona estos niveles y las
   * prioridades software asignadas a cada uno de ellos
   *
   * nivel enum   CIP  Mascara Prioridades Dispositivo
   * 0     Cero   IRQ0  bit 0   255 - 252   Despertador del sistema
   * 1     Uno    IRQ1  bit 1   251 - 248   Teclado
   * 2     Dos    IRQ2  bit 2   247 - 244
   * 3     Tres   IRQ3  bit 3   243 - 240   Mouse
   * 4     Cuatro IRQ4  bit 4   239 - 236   COM1
   * 5     Cinco  IRQ5  bit 5   235 - 232
```

```
   * 6     Seis   IRQ6  bit 6   231 - 228   USB
   * 7     Siete  IRQ7  bit 7   227 - 224   LPT
   * 8 - 15 .....................223 - 192
   *
   * El clasificdor Nivel del tipo enum, definido a continua-
   * ción, se usa para el manejo de estos niveles*/

  enum Nivel
  {
    Cero, Uno, Dos, Tres, Cuatro, Cinco, Seis, Siete
  }

  public partial class MascaraInt
  {
    //longitud (niveles o bits) de la mascara de interrupciones
    private int niveles = Enum.GetNames(typeof(Nivel)).Length;

    //palabra de control para deshabilitar interrupcion hardware
    private BitArray finDeInt;

    //mascara programable: arreglo binario de "niveles" bits
    private BitArray mascara;

    protected MascaraInt()
    {
      //mascara de "niveles" bits, cada uno con el valor false == "0"
      mascara = new BitArray(niveles, false);

      //Establecer "finDeInt" para el CIP =  "00000010"
      finDeInt = new BitArray(niveles, false);
      finDeInt.Set(1, true);
   }
  }
}
```

////Componente:MascaraInt2.cs

```
using System.Collections;

namespace GASP
{
  partial class MascaraInt
  {
    internal BitArray Mascara
    {
      get
      {
        return mascara;
      }
      set
      {
        mascara = value;
      }
    }

    internal BitArray MascaraInicial
    {
      get
      {
        //configuración incial de la mascara
        BitArray mascaraIni = new BitArray(8, false);
        mascaraIni.Set(1, true); //teclado habilitado
```

```
            mascaraIni.Set(3, true); //Mouse habilitado
            mascaraIni.Set(6, true); //USB habilitado
            return mascaraIni;
        }
      }

      internal BitArray FinDeInt
      {
        get
        {
          return finDeInt;
        }
      }

      public int Niveles
      {
        get
        {
          return niveles;
        }
      }

      //prioridades soft por nivel hard
      public int PrioridadesXnivel
      {
        get
        {
          return 4;
        }
      }

      //maxima prioridad
      public int MaxP
      {
        get
        {
          return 255;
        }
      }
    }
}
```

///Paquete: Hard – emulador de hardware

////Componente: BaseTiempos.cs

```
using System;

namespace GASP
{
  delegate void GeneradorReloj();
  public class BaseTiempos
  {
    //BASE DE TIEMPOS

    //reloj basico de funcionamiento del entorno operativo
    //dado por periodo en milisegundos para el Timer
    private int periodo;
    //contador de periodos
    private int contadorPeriodos;
    //unidad de tiempo, fundamental considerada por el sistema
```

```
    private int periodosXUT;

    //temporizador
    private System.Timers.Timer tiempo;

    private event GeneradorInterrupcion evIRQ0;
    private event GeneradorReloj evReloj;

    internal BaseTiempos()
    {
      periodo = 600;
      contadorPeriodos = 0;
      periodosXUT = 8;
      tiempo = new System.Timers.Timer(periodo);
      tiempo.Elapsed += new
System.Timers.ElapsedEventHandler(frecuencia);
    }

    private void frecuencia(object source,
System.Timers.ElapsedEventArgs e)
    {
      contadorPeriodos++;

      if (contadorPeriodos == periodosXUT)
      {
        contadorPeriodos = 0;
        if (evIRQ0 != null)
        {
          evIRQ0();
          return;
        }
      }

      if (evReloj != null)
        evReloj();
    }

    internal GeneradorInterrupcion IRQ0
    {
      set
      {
        evIRQ0 += value;
      }
    }

    internal GeneradorReloj reloj
    {
      set
      {
        evReloj += value;
      }
    }

    public bool Habilitado
    {
      get
      {
        return tiempo.Enabled;
      }
      set
      {
```

```
        tiempo.Enabled = value;
      }
    }

    public int Periodo
    {
      get
      {
        return periodo;
      }
      set
      {
        periodo = value < 600 ? 600 : value;
        tiempo.Interval = periodo;
      }
    }

    public int PeriodosXUT
    {
      get
      {
        return periodosXUT;
      }
      set
      {
        periodosXUT = value < 8 ? 8 : value;
      }
    }
   }
}
```

////Componente: CIP.cs

```
using System;
using System.Collections;
using System.Text;

namespace GASP
{
  enum dirES
  {
    cip00, //direccion para finalizar solicitud de interrupcion
    cip01  //direccion para manejar mascara de interrupciones
  }

  //Controlador de Interrupciones Programable
  class CIP : MascaraInt
  {
    //registro de "mascara" del CIP, se hereda de MascaraInt

    //registro para "fin de interrupcion" del CIP
    private BitArray unRegControl;

    internal CIP()
    {
      unRegControl = new BitArray(Niveles, false);
    }

    internal BitArray this[dirES dir]
    {
      set
      {
```

```
        if (dir == dirES.cip01)
        { //preparacion de mecanismo de interrupcion: habil I/O
          unRegControl.SetAll(false);
          Mascara.SetAll(false);
          Mascara.Or(value);
        }
        else
          //señalizacion de Fin de Interrupcion al CIP
          unRegControl.Or(value);
      }
      get
      {
        if (dir == dirES.cip01)
          return Mascara;
        else
          return unRegControl;
      }
    }
  }
}
```

////Componente: Interrupcion.cs

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;

namespace GASP
{
  delegate void GeneradorInterrupcion();
  delegate void ManejadorInterrupcion(Nivel nivel);
  class Interrupcion
  {
      /*I N T E R R U P C I O N E S controladas por el CIP
       * lo simulamos con eventos externos, cada tipo de
       * los cuales se representa por una instancia de
       * la clase "Interrupcion":
       *
       * hard: deshabilita interrupciones y salva el PC
       * asm: salva registros hard
       *
       * verifica que la interrupcion este habilitada
       * "mascaraCIP[(int)nivel] == true" y traslada el
       * control al servidor de interrupciones.*/

    private Nivel nivel; //tipo de interrupcion
    private BitArray mascaraCIP;
    public event ManejadorInterrupcion  Servidor;

    internal Interrupcion(Nivel niv, BitArray masca)
    {
      nivel = niv;
      mascaraCIP = masca;
    }
    internal void Interrumpir()
    {
      if (mascaraCIP[(int)nivel])
      {
        /* enpilar contador de programa y palabra de estado
         * enpilar otros registros (por soft)
         */
```

```
        if (Servidor != null)
        {
          Servidor(nivel);
        }
      }
    }
  }
}
```

////Componente: MemCompartida.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  public interface IValor
  {
    int Valor { get;set;}
  }

  public class MemCompartida: IValor
  {
    int valor = 0;
    public event VerBufferUno Mostrar;

    public MemCompartida()
    {
      valor = 0;
    }

    object vista;
    public object Vista
    {
      get { return vista; }
      set { vista = value; }
    }

    public int Valor
    {
      get
      {
        if(Mostrar != null)
          Mostrar("L", Thread.CurrentThread.Name, valor);

        return valor;
      }
      set
      {
        valor = value;

        if(Mostrar != null)
          Mostrar("E", Thread.CurrentThread.Name, valor);
      }
    }
  }
}
```

////Componente: BufferUno.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  public delegate void VerBufferUno(string mov, string nomSubP, int
mem);

  public class BufferUno : IValor
  {
    private int memoria = -1;

    public event VerBufferUno Mostrar;

    public void Poner(int dato)
    {
      if(Mostrar != null)
        Mostrar("P", Thread.CurrentThread.Name, dato);

      memoria = dato;
    }

    public int Sacar()
    {
      if(Mostrar!= null)
        Mostrar("C", Thread.CurrentThread.Name, memoria);

      return memoria;
    }

    //Inf.adicional para manejo de visualizacion
    public int Valor
    {
      get {return memoria; }
      set {memoria = value;}
    }

    private object vista;
    public object Vista
    {
      get {return vista; }
      set {vista = value;}
    }
  }
}
```

////Componente: BufferLim.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  public delegate void VerBufferX(string mov, string nomSubP, int
dato, int indice, int elementos);

  public class BufferLim : IValor
  {
```

```
    private int[] buffer;
    private int longitud, elementos, indiceE, indiceL;

    public event VerBufferX Mostrar;

    public BufferLim()
    {
      longitud = 3;
      buffer = new int[longitud];
      indiceE = 0;
      indiceL = 0;
      elementos = 0;
    }

    public void Poner(int dato)
    {
      buffer[indiceE] = dato;
      elementos++;

      if (Mostrar != null)
        Mostrar("P", Thread.CurrentThread.Name, dato, indiceE,
elementos);

      indiceE = ++indiceE % longitud;
    }

    public int Sacar()
    {
      //VerCodigo("int temp = memoria\n");
      int temp = buffer[indiceL];
      elementos--;

      //VerCodigo("if(Mostrar != null) ==>\n");
      if (Mostrar != null)
        Mostrar("C", Thread.CurrentThread.Name, temp, indiceL,
elementos);

      indiceL = ++indiceL % longitud;

      //VerCodigo("return temp\n");
      return temp;
    }

    //Inf.adicional para manejo de visualizacion
    public int Valor
    {
      get {return longitud; }
      set
      {
        if (longitud > 0 && longitud < 100 && longitud != value)
        {
          longitud = value;
          buffer = new int[longitud];
          indiceE = 0;
          indiceL = 0;
          elementos = 0;
          if (Mostrar != null)
            Mostrar("I", Thread.CurrentThread.Name, 0, 0, longitud);
        }
      }
    }
```

```
    public object[] ValorActual
    {
      get
      {
        object[] objs = new object[longitud];
        for (int obj = 0; obj < longitud; obj++)
          objs[obj] = buffer[obj];
        return objs;
      }
    }

    private object vista;
    public object Vista
    {
      get {return vista; }
      set {vista = value;}
    }
  }
}
```

////Componente: BufferIli.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  public delegate void VerBuffer(string mov, string nomSubP, int dato,
int indice, int elementos);

  public class BufferIli : IValor
  {
    private ArrayList buffer;
    private int elementos, indiceE, indiceL;

    public event VerBuffer Mostrar;

    public BufferIli()
    {
      buffer = new ArrayList();
      indiceE = 0;
      indiceL = 0;
      elementos = 0;
    }

    public void Poner(int dato)
    {
      buffer.Add(dato);
      elementos++;

      if (Mostrar != null)
        Mostrar("P", Thread.CurrentThread.Name, dato, indiceE,
elementos);

      indiceE++;
    }

    public int Sacar()
```

```
    {
      //VerCodigo("int temp = memoria\n");
      int temp = (int)buffer[0];
      buffer.RemoveAt(0);
      elementos--;

      //VerCodigo("if(Mostrar != null) ==>\n");
      if (Mostrar != null)
        Mostrar("C", Thread.CurrentThread.Name, temp, indiceL,
elementos);

      indiceL++;

      //VerCodigo("return temp\n");
      return temp;
    }

    //Inf.adicional para manejo de visualizacion
    public int Valor
    {
      get { return elementos; }
      set
      { }
    }

    public object[] ValorActual
    {
      get
      {
       return buffer.ToArray();
      }
    }

    private object vista;
    public object Vista
    {
      get { return vista; }
      set { vista = value; }
    }
  }
}
```

///Paquete: EEBB – elementos básicos

////Componente: BCP1.cs

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;
using System.Threading;

namespace GASP
{
  enum ProcesoEstado
  {
    Creado, EnEjecucion, Listo, Dormido, Esperando, Expropiado,
Transitorio, Abortado, Especial
  }

  enum ProcesoTipo
```

```
  {
    Reactivo, Interactivo, Trabajador, Especial
  }

  public partial class BCP : MascaraInt
  {
    private Thread procesoId;
    private AutoResetEvent reloj;
    private string nombre;
    private int prioridad;
    private ProcesoEstado estado;
    private ProcesoTipo tipo;
    private int dormirCuota;
    private Mensaje mensaje;
    private Buzon espera;
    private BCP sigDormido;
    private BCP sigEspera;
    private BCP sigListo;
    private BCP siguiente;
    static private int todos;

    /*  referencias de pila
     *  referencias de memoria
     *  referencias de archivos
     *  datos administrativos y estadisticos
     */

    static BCP()
    {
      todos = 0;
    }
  }
}
```

////Componente: BCP2.cs

```
using System.Threading;

namespace GASP
{
  partial class BCP
  {
    //Constructor para crear las cabeceras de las colas:
    //cab = 1, cola "listos" que tiene maxima prioridad: 255
    //cab = 2, cola "dormidos" que tiene cuota para dormir = 0

    internal BCP(int cab)
    {
      procesoId = null;
      reloj = null;
      nombre = cab == 1 ? "CabListo" : "CabDormi";
      prioridad = cab == 1 ? 255 : 0;
      estado = ProcesoEstado.Especial;
      tipo = ProcesoTipo.Especial;
      dormirCuota = 0;
      mensaje = null;
      espera = null;
      sigDormido = sigEspera = sigListo = null;
      siguiente = null;
    }

    /* constructor normal de acceso restringido
```

```
     */
    internal BCP(Thread proc, int prio, string nomb, BCP sig)
    {
      todos++;
      procesoId = proc;
      reloj = new AutoResetEvent(false);
      nombre = nomb;

      derivarPrioridad(prio);

      estado = ProcesoEstado.Creado;
      dormirCuota = 0;
      mensaje = null;
      espera = null;
      sigDormido = sigEspera = sigListo = null;
      siguiente = sig;
    }

    private void derivarPrioridad(int prio)
    {
      // maxima prioridad = MaxP
      //acotacion de prioridad de proceso
      prioridad = prio > MaxP ? MaxP : prio;
      prioridad = prioridad < 0 ? 0 : prioridad;

      //re/definir tipo de proceso
      //maxima prioridad  solo soft, no maneja interrups
      int maxPsoft = MaxP - Niveles * PrioridadesXnivel;
      if (prioridad > maxPsoft)
        tipo = ProcesoTipo.Reactivo;
      else if (prioridad > (maxPsoft / 2)) //entre 223 y 112
        tipo = ProcesoTipo.Interactivo;
      else tipo = ProcesoTipo.Trabajador; //entre 0 y 111

      //re/definir mascara de interrupciones de proceso
      if (prioridad <= maxPsoft)
         //todas las interrupciones habilitadas
        Mascara.SetAll(true);
      else
      {
        //poner a 1 los bits con mayor prioridad
        int nivel = (MaxP - prioridad) / PrioridadesXnivel;
        for (int i = 0; i < nivel; i++)
          Mascara.Set(i, true);
      }
    }
  }
}
```

////Componente: BCP3.cs

```
using System.Threading;

namespace GASP
{
  partial class BCP
  {
    internal Thread ProcesoId
    {
      get
      {
        return procesoId;
```

```
    }
  }

  public string Nombre
  {
    get
    {
      return nombre;
    }
    set
    {
      nombre = value;
    }
  }

  internal int Prioridad
  {
    get
    {
      return prioridad;
    }
    set
    {
      derivarPrioridad(value);
    }
  }

  internal ProcesoTipo Tipo
  {
    get
    {
      return tipo;
    }
  }

  internal ProcesoEstado Estado
  {
    get
    {
      return estado;
    }
    set
    {
      estado = value;
    }
  }

  internal Mensaje Mensaje
  {
    get
    {
      return mensaje;
    }
    set
    {
      mensaje = value;
    }
  }

  internal Buzon Espera
  {
```

```
    get
    {
      return espera;
    }
    set
    {
      espera = value;
    }
  }

  internal BCP SigDormido
  {
    get
    {
      return sigDormido;
    }
    set
    {
      sigDormido = value;
    }
  }

  internal BCP SigEspera
  {
    get
    {
      return sigEspera;
    }
    set
    {
      sigEspera= value;
    }
  }

  internal BCP SigListo
  {
    get
    {
      return sigListo;
    }
    set
    {
      sigListo = value;
    }
  }

  internal BCP Siguiente
  {
    get
    {
      return siguiente;
    }
    set
    {
      siguiente = value;
    }
  }

  static public int Todos
  {
    get
```

```
      {
        return todos;
      }
    }
  }
}
```

////Componente: BCP4.cs

```
using System.Threading;

namespace GASP
{
  partial class BCP
  {
    public AutoResetEvent Reloj
    {
      get
      {
        return reloj;
      }

      internal set
      {
        reloj = value;
      }
    }

    public int DormirCuota
    {
      get
      {
        return dormirCuota;
      }

      set
      {
        dormirCuota = value;
      }
    }

    public void EsperarReloj()
    {
      reloj.WaitOne();
    }
  }
}
```

////Componente: Buzon1.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  public partial class Buzon
  {
    static private int generadorId = 0;
    private int buzonId;
    private Mensaje primerMensaje;
```

```
private Mensaje ultimoMensaje;
private BCP primerProceso;
private BCP ultimoProceso;
private object vista;
private Buzon siguiente;

internal Buzon(Buzon sig)
{
  buzonId = generadorId++;
  primerMensaje = ultimoMensaje = null;
  primerProceso = ultimoProceso = null;
  vista = null;
  siguiente = sig;
}

public int BuzonId
{
  get
  {
    return buzonId;
  }
}

public object Vista
{
  get
  {
    return vista;
  }
  set
  {
    vista = value;
  }
}

internal Buzon Siguiente
{
  get
  {
    return siguiente;
  }
  set
  {
    siguiente = value;
  }
}

internal BCP PrimerProceso
{
  get
  {
    return primerProceso;
  }
}

internal Mensaje PrimerMensaje
{
  get
  {
    return primerMensaje;
  }
```

```
    }
  }
}
```

////Componente: Buzon2Mensajes.cs

```
using System.Threading;

namespace GASP
{
  public delegate void delAlBuzon(string s);
  public delegate void delForzarBuzon(string s);
  public delegate void delDelBuzon();

  partial class Buzon
  {
    public event delAlBuzon evAlBuzonP;
    public event delDelBuzon evDelBuzonM;
    public event delForzarBuzon evForzarBuzonM;

    internal bool HayMensajeEsperando()
    {
      return primerMensaje != null;
    }

      internal void InsertarMensajeEsperando(Mensaje mensaje)
      {
        Monitor.Enter(this);
        if (primerMensaje == null)
          primerMensaje = mensaje;
        else
          ultimoMensaje.Siguiente = mensaje;

        ultimoMensaje = mensaje;

        //marcar mensaje producido
        mensaje.Siguiente = null;

        //avizar a interfase
        if (evAlBuzonM != null)
        {
          evAlBuzonM(mensajeToString(mensaje));
        }
        Monitor.Exit(this);
      }

    internal Mensaje EliminarMensajeEsperando()
    {
      Monitor.Enter(this);
      Mensaje mensaje = primerMensaje;
      primerMensaje = mensaje.Siguiente;

      if (primerMensaje == null)
        ultimoMensaje = null;

      //marcar mensaje consumido
      mensaje.Siguiente = mensaje;

      //avisar a interfase
      if (evDelBuzonM != null)
        evDelBuzonM();
      Monitor.Exit(this);
```

```
      return mensaje;
    }

    internal void EliminarMensajeEsperando(Mensaje mensaje)
    {
      Monitor.Enter(this);
      Mensaje anterior, actual;
      anterior = actual = primerMensaje;

      while (mensaje != actual)
      {
        anterior = actual;
        actual = actual.Siguiente;
      }

      if (actual == anterior)
        primerMensaje = actual.Siguiente;
      else
        anterior.Siguiente = actual.Siguiente;

      if (mensaje.Siguiente == null)
        ultimoMensaje = (primerMensaje == null ? null : anterior);

      //avisar interfase
      if (evForzarBuzonM != null)
        evForzarBuzonM(mensajeToString(mensaje));
      Monitor.Exit(this);
    }
  }
}
```

////Componente: Buzon3MensajesLeer.cs

```
using System.Collections;
using System.Threading;

namespace GASP
{
  partial class Buzon
  {
    internal ArrayList LeerMensajes()
    {
      Monitor.Enter(this);
      ArrayList mLista = new ArrayList();
      Mensaje actual = primerMensaje;
      while (actual != null)
      {
        mLista.Add(mensajeToString(actual));
        actual = actual.Siguiente;
      }
      Monitor.Exit(this);
      return mLista;
    }
  }
}
```

////Componente: Buzon4MjeACadena.cs

```
namespace GASP
{
  partial class Buzon
  {
```

```
    private string mensajeToString(Mensaje mensaje)
    {
      string sMensaje;
      switch (mensaje.Tipo)
      {
        case MensajeTipo.Normal:
          if (mensaje.Cuerpo != null)
            sMensaje = "N:" + mensaje.Cuerpo;
          else
            sMensaje = "SYNC";
          break;
        case MensajeTipo.Interrupcion:
          sMensaje = "Interrupcion";
          break;
        case MensajeTipo.Perdido:
          sMensaje = "Perdido";
          break;
        default:
          sMensaje = "...???...";
          break;
      }

      return sMensaje;
    }
  }
}
```

////Componente: Buzon5Procesos.cs

```
using System.Threading;

namespace GASP
{

  partial class Buzon
  {
    public event delDelBuzon evDelBuzonP;
    public event delAlBuzon evAlBuzonM;
    public event delForzarBuzon evForzarBuzonP;

    internal bool HayProcesoEsperando()
    {
      return primerProceso != null;
    }

    internal void InsertarProcesoEsperando(BCP proceso)
    {
      Monitor.Enter(this);
      if (ultimoProceso == null)
        primerProceso = proceso;
      else
        ultimoProceso.SigEspera = proceso;

      ultimoProceso = proceso;

      proceso.Estado = ProcesoEstado.Esperando;
      proceso.Espera = this;
      proceso.SigEspera = null;

      //avisar a interfase
      if (evAlBuzonP != null)
        evAlBuzonP(proceso.Nombre);
```

```
      Monitor.Exit(this);
    }

    internal BCP EliminarProcesoEsperando()
    {
      Monitor.Enter(this);
      BCP proceso = primerProceso;
      proceso.Estado = ProcesoEstado.Transitorio;
      primerProceso = proceso.SigEspera;
      proceso.SigEspera = null;
      proceso.Espera = null;

      if (primerProceso == null)
        ultimoProceso = null;

      //marcar mensaje consumido
      //mensaje.Siguiente = mensaje;
      //avizar a interfase
      if (evDelBuzonP != null)
        evDelBuzonP();
      Monitor.Exit(this);
      return proceso;
    }

    internal void EliminarProcesoEsperando(BCP proceso)
    {
      Monitor.Enter(this);
      BCP anterior, actual;
      anterior = actual = primerProceso;

      while (proceso != actual)
      {
        anterior = actual;
        actual = actual.SigEspera;
      }

      proceso.Estado = ProcesoEstado.Transitorio;
      if (actual == anterior)
        primerProceso = actual.SigEspera;
      else
        anterior.SigEspera = actual.SigEspera;

      if (proceso.SigEspera == null)
        ultimoProceso = (primerProceso == null ? null : anterior);

      proceso.SigEspera = null;
      proceso.Espera = null;

      //avisar interfase
      if (evForzarBuzonP != null)
        evForzarBuzonP(proceso.Nombre);
      Monitor.Exit(this);
    }
  }
}
```

////Componente: Buzon6ProcesosLeer.cs

```
using System.Collections;
using System.Threading;

namespace GASP
```

```
{
  partial class Buzon
  {
    internal ArrayList LeerProcesos()
    {
      Monitor.Enter(this);
      ArrayList mLista = new ArrayList();
      BCP actual = primerProceso;
      while (actual != null)
      {
        mLista.Add(actual.Nombre);
        actual = actual.SigEspera;
      }
      Monitor.Exit(this);
      return mLista;
    }
  }
}
```

////Componente: Mensaje1.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace GASP
{
  internal enum MensajeTipo
  {
    Normal, Interrupcion, Perdido
  }

  public partial class Mensaje
  {
    //cuando el elemento siguiente apunta al mismo mensaje,
    //éste esta consumido o inactivo
    private Mensaje siguiente;
    private MensajeTipo tipo;
    private object cuerpo;


    Mensaje(Mensaje sig, MensajeTipo tip, object cuer)
    {
      siguiente = sig;
      tipo = tip;
      cuerpo = cuer;
    }

    internal Mensaje Siguiente
    {
      get
      {
        return siguiente;
      }

      set
      {
        siguiente = value;
      }
    }

    internal MensajeTipo Tipo
```

```
    {
      get
      {
        return tipo;
      }

      set
      {
        tipo = value;
      }
    }
  }
}
```

////Componente: Mensaje2.cs

```
namespace GASP
{
  partial class Mensaje
  {
    public Mensaje()
      : this(null, MensajeTipo.Normal, null)
    { }

    public Mensaje(object cuerpoRef)
      : this(null, MensajeTipo.Normal, cuerpoRef)
    { }

    public Mensaje(Mensaje mje)
    {
      siguiente = mje.siguiente;
      tipo      = mje.tipo;
      cuerpo    = mje.cuerpo;
    }

    public object Cuerpo
    {
      get
      {
        return cuerpo;
      }
      set
      {
        cuerpo = value;
      }
    }
  }
}
```

///Paquete: Listas – listas de elementos básicos

////Componente: Listas1.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace GASP
{
  public partial class Listas : MascaraInt
  {
    private BCP enEjecucion;
```

```
    private BCP procesos;
    private BCP listos;
    private BCP dormidos;
    private Buzon buzones;
    private string proNombre, lisNombre, dorNombre, buzNombre;

    internal Listas()
      : this("procesos", "listos", "dormidos", "buzones")
    { }

    internal Listas(string proc, string list, string dorm, string
buzo)
    {
      enEjecucion = procesos = listos = dormidos = null;
      buzones = null;
      proNombre = proc;
      lisNombre = list;
      dorNombre = dorm;
      buzNombre = buzo;
      iniciarListosDormidos();
    }

    private void iniciarListosDormidos()
    {
      //crea cabecera de lista procesos "listos"
      listos = new BCP(1);

      //crea cabecera de lista procesos "dormidos"
      dormidos = new BCP(2);
    }

    internal BCP EnEjecucion
    {
      get
      {
        return enEjecucion;
      }
      set
      {
        enEjecucion = value;
      }
    }
  }
}
```

////Componente: Listas2BCPs.cs

```
using System.Collections;
using System.Threading;

namespace GASP
{
  internal delegate void delProcesoCreado(string[] datos);
  internal delegate void delProcesoEliminado(string nombre);
  public delegate void delProcesoAbortado(string nombre);

  partial class Listas
  {
    internal delProcesoCreado evProCreado;
    internal delProcesoEliminado evProEliminado;
    public delProcesoAbortado evProAbortado;
```

```
    internal string ProNombre
    {
      get
      {
        return proNombre;
      }
    }

    internal BCP Procesos
    {
      get
      {
        return procesos;
      }
    }

    //metodo para insertar BCP´s

    internal BCP InsertarProceso(Thread procId, int prio, string nomb)
    {
      lock (this)
      {
        procesos = new BCP(procId, prio, nomb, procesos);
      }

      //avisar consola
      if (verElProceso && elProcesoNombre == nomb)
        mostrarEstado("Creado");

      //avisar interfaz de subProcesos
      if (evProCreado != null)
      {
        string[] s =
{nomb,procesos.Prioridad.ToString(),procesos.Tipo.ToString()};

        evProCreado(s);
      }

      return procesos;
    }

    internal void EliminarProceso(BCP proceso)
    {
      BCP anterior, actual;
      anterior = actual = procesos;

      Monitor.Enter(this);
      while (proceso != actual)
      {
        anterior = actual;
        actual = actual.Siguiente;
      }

      if (actual == anterior)
        procesos = procesos.Siguiente;
      else
        anterior.Siguiente = actual.Siguiente;

      //avisar a interfase
      //consola
      if (verElProceso && proceso.Nombre == elProcesoNombre)
```

```
        if (proceso.Estado != ProcesoEstado.Abortado)
          mostrarEstado("Finalizado");
        else mostrarEstado("Abortado");


      if (evProEliminado != null && proceso.Estado !=
ProcesoEstado.Abortado)
        //evento proceso eliminado
        evProEliminado(proceso.Nombre);
      else if (evProAbortado != null && proceso.Estado ==
ProcesoEstado.Abortado)
        //evento proceso abortado;
        evProAbortado(proceso.Nombre);

      Monitor.Exit(this);
    }

    public BCP BuscarProceso(string nombre)
    {
      Monitor.Enter(this);
      BCP anterior, actual, proceso = null;
      anterior = actual = procesos;

      while (nombre != actual.Nombre && actual.Siguiente != null)
      {
        anterior = actual;
        actual = actual.Siguiente;
      }

      if (nombre == actual.Nombre)
        proceso = actual;

      Monitor.Exit(this);
      return proceso;
    }

    internal void DisponerProcesos()
    {
      BCP proceso = procesos;
      Monitor.Enter(this);
      while (proceso != null)
      {
        proceso.ProcesoId.Abort();
        proceso = proceso.Siguiente;
      }
      Monitor.Exit(this);
    }

    public ArrayList ListarBCPs()
    {
      BCP proceso = procesos;
      ArrayList alPP = new ArrayList();

      Monitor.Enter(this);
      while (proceso != null)
      {
        alPP.Add(proceso.Nombre + "," +
          proceso.Prioridad.ToString() + "," +
          //proceso.Estado.ToString() + "," +
          proceso.Tipo.ToString());
```

```
        proceso = proceso.Siguiente;
      }
      Monitor.Exit(this);

      return alPP;
    }
  }
}
```

////Componente: Listas3Listos.cs

```
using System.Threading;
namespace GASP
{
  partial class Listas
  {
    internal string LisNombre
    {
      get
      {
        return lisNombre;
      }
    }

    internal BCP Listos
    {
      get
      {
        return listos.SigListo;
      }
    }

    internal void InsertarListo(BCP proceso)
    {
      BCP anterior, actual;
      anterior = actual = listos;
      Monitor.Enter(this);

      while (proceso.Prioridad <= actual.Prioridad && actual.SigListo
!= null)
      {
        anterior = actual;
        actual = actual.SigListo;
      }

      proceso.SigListo = anterior.SigListo;
      anterior.SigListo = proceso;
      proceso.Estado = ProcesoEstado.Listo;

      if (verElProceso && elProcesoNombre == proceso.Nombre)
        mostrarEstado("Listo");

      Monitor.Exit(this);
    }

    //eliminar proceso en ejecucion de la cola "listos"
    internal BCP EliminarListo()
    {
      BCP anterior, actual, proceso = enEjecucion;
      anterior = actual = listos;
      Monitor.Enter(this);
```

```
      proceso.Estado = ProcesoEstado.Transitorio;
      while (proceso != actual)
      {
        anterior = actual;
        actual = actual.SigListo;
      }

      //omision = eliminacion
      anterior.SigListo = actual.SigListo;
      proceso.SigListo = null;
      Monitor.Exit(this);
      return proceso;
    }

    internal void EliminarListo(BCP proceso)
    {
      BCP anterior, actual;
      anterior = actual = listos;
      Monitor.Enter(this);

      while (actual.SigListo != null && proceso != actual)
      {
        anterior = actual;
        actual = actual.SigListo;
      }

      if (proceso == actual)
      {
        proceso.Estado = ProcesoEstado.Transitorio;
        anterior.SigListo = actual.SigListo;
        proceso.SigListo = null;
      }
      Monitor.Exit(this);
    }
  }
}
```

////Componente: Listas4Dormidos.cs

```
using System.Threading;

namespace GASP
{
  partial class Listas
  {
    internal string DorNombre
    {
      get
      {
        return dorNombre;
      }
    }

    internal BCP Dormidos
    {
      get
      {
        return dormidos.SigDormido;
      }
    }

    /*Pasar proceso en ejecucion a la lista de Dormidos*/
```

```
    internal void InsertarDormido(int uuTiempo)
    {
      BCP proceso, actual, anterior;

      Monitor.Enter(this);

      proceso = enEjecucion;
      anterior = actual = dormidos;
      while (uuTiempo >= actual.DormirCuota && actual.SigDormido !=
null)
      {
        uuTiempo -= actual.DormirCuota;
        anterior = actual;
        actual = actual.SigDormido;
      }

      if (uuTiempo < actual.DormirCuota)
      {
        actual.DormirCuota -= uuTiempo;
        actual = anterior;
      }
      else
        uuTiempo -= actual.DormirCuota;

      proceso.SigDormido = actual.SigDormido;
      actual.SigDormido = proceso;
      proceso.DormirCuota = uuTiempo;
      proceso.Estado = ProcesoEstado.Dormido;

      if (verElProceso && elProcesoNombre == proceso.Nombre)
        mostrarEstado("Dormido");

      Monitor.Exit(this);
    }

    internal BCP EliminarDormido()
    {
      BCP proceso = dormidos.SigDormido;
      Monitor.Enter(this);
      proceso.Estado = ProcesoEstado.Transitorio;
      dormidos.SigDormido = proceso.SigDormido;
      proceso.SigDormido = null;
      Monitor.Exit(this);
      return proceso;
    }

    internal void EliminarDormido(BCP proceso)
    {
      BCP anterior, actual;
      anterior = actual = dormidos;
      Monitor.Enter(this);

      while (actual.SigDormido != null && proceso != actual)
      {
        anterior = actual;
        actual = actual.SigDormido;
      }

      if (proceso == actual)
      {
```

```
        proceso.Estado = ProcesoEstado.Transitorio;
        anterior.SigDormido = actual.SigDormido;
        if (proceso.SigDormido != null)
          anterior.SigDormido.DormirCuota += proceso.DormirCuota;

        proceso.SigDormido = null;
      }

      Monitor.Exit(this);
    }
  }
}
```

////Componente: Listas5AbortarProceso.cs

```
using System.Threading;

namespace GASP
{
  partial class Listas
  {
    internal bool AbortarProceso(BCP proceso)
    {
      Monitor.Enter(this);
      bool flagEnEjecucion = false;
      proceso.ProcesoId.Abort();
      switch (proceso.Estado)
      {
        case ProcesoEstado.Dormido:
          EliminarDormido(proceso);
          break;
        case ProcesoEstado.Esperando:
          proceso.Espera.EliminarProcesoEsperando(proceso);
          break;
        default:
          if (proceso.Estado == ProcesoEstado.EnEjecucion)
            flagEnEjecucion = true;

          EliminarListo(proceso);
          break;
      }

      proceso.Estado = ProcesoEstado.Abortado;
      EliminarProceso(proceso);

      Monitor.Exit(this);
      return flagEnEjecucion;
    }
  }
}
```

////Componente: Listas6Monitorear.cs

```
using System.Text;

namespace GASP
{

  public delegate void ManejadorConsola(string s);

  partial class Listas
  {
```

```
    //MANEJO DE CONSOLA DEL SISTEMA: 7 variables, 5 propiedades, 7
metodos
    //evento al que se vincula la consola
    public event ManejadorConsola Remitir;
    //ver evolucion de "el proceso"
    protected bool verElProceso = false;
    protected string elProcesoNombre = "sListo";

    //ver cambio de modo: (usuario)>>>GPPE y (usuario)<<<GPPE
    protected bool verCambioModo = false;

    //ver Conmutacion conmutacion de proceso
    protected bool verConmutacion = false;

    //ver interrupcion y proceso interrumpido
    protected bool verInterrupcion = false;

    //mensaje de evolucion de proceso diferido
    private string mjeDiferido = null;

    public string ElProcesoNombre
    {
      get
      {
        return elProcesoNombre;
      }
      set
      {
        elProcesoNombre = value;
      }
    }

    public bool VerElProceso
    {
      get
      {
        return verElProceso;
      }
      set
      {
        verElProceso = value;
      }
    }

    public bool VerCambioModo
    {
      get
      {
        return verCambioModo;
      }
      set
      {
        verCambioModo = value;
      }
    }

    public bool VerConmutacion
    {
      get
      {
        return verConmutacion;
```

```
    }
    set
    {
      verConmutacion = value;
    }
  }

  public bool VerInterrupcion
  {
    get
    {
      return verInterrupcion;
    }
    set
    {
      verInterrupcion = value;
    }
  }

  protected void mostrarEstado(string estado)
  {
    if (Remitir != null)
      if (!verConmutacion)
        Remitir(elProcesoNombre + ":" + estado + "\n");
      else
        mjeDiferido = elProcesoNombre + ":" + estado + "\n";
  }

  protected void ingresarGPPE(string funcion)
  {
    if(Remitir != null)
      Remitir(EnEjecucion.Nombre + " >>> GPPE:" + funcion + "\n");
  }

  protected void ingresarGPPE(string nombre, string funcion)
  {
    if (Remitir != null)
      Remitir(nombre + " >>> GPPE:" + funcion + "\n");
  }

  protected void salirGPPE()
  {
    if (Remitir != null)
      Remitir(EnEjecucion.Nombre + " <<< GPPE\n");
  }

  protected void mostrarDeProceso()
  {
    if (Remitir != null)
    {
      if (EnEjecucion != null)
        Remitir(EnEjecucion.Nombre + " >>=a=>> ");
      else
        Remitir("Primer proceso: ");
    }
  }

  protected void mostrarAProceso()
  {
    if (Remitir != null)
    {
```

```
        Remitir(EnEjecucion.Nombre + "\n");
        if (verElProceso)
          Remitir(mjeDiferido);
      }
    }

    internal void mostrarInterrupcion(Nivel nivel)
    {
      if (Remitir != null)
      {
        string sTemp;
        switch (nivel)
        {
          case Nivel.Cero: sTemp = "IRQ0"; break;
          case Nivel.Uno: sTemp = "IRQ1"; break;
          case Nivel.Dos: sTemp = "IRQ2"; break;
          case Nivel.Tres: sTemp = "IRQ3"; break;
          case Nivel.Cuatro: sTemp = "IRQ4"; break;
          case Nivel.Cinco: sTemp = "IRQ5"; break;
          case Nivel.Seis: sTemp = "IRQ6"; break;
          case Nivel.Siete: sTemp = "IRQ7"; break;
          default: sTemp = "????"; break;
        }
        Remitir(sTemp + " interrumpio " + EnEjecucion.Nombre + "\n");
      }
    }
  }
}
```

////Componente: Listas7Buzones.cs

```
using System.Collections;
using System.Threading;

namespace GASP
{
  partial class Listas
  {
    internal string BuzNombre
    {
      get
      {
        return buzNombre;
      }
    }

    internal Buzon Buzones
    {
      get
      {
        return buzones;
      }
    }

    internal Buzon InsertarBuzon()
    {
      Buzon buzon;
      lock (this)
      {
        buzones = buzon = new Buzon(buzones);
      }
      return buzon;
```

```
}

internal bool EliminarBuzon(Buzon buzo)
{
  bool flagIndicador = false;
  Buzon anterior, actual;
  Monitor.Enter(this);

  anterior = actual = buzones;

  while (buzo != actual && actual.Siguiente != null)
  {
    anterior = actual;
    actual = actual.Siguiente;
  }

  if (buzo == actual && buzo.PrimerProceso == null)
  {
    if (actual == anterior)
      buzones = buzones.Siguiente;
    else
      anterior.Siguiente = actual.Siguiente;

    flagIndicador = true;
  }
  Monitor.Exit(this);
  return flagIndicador;
}

internal bool EliminarBuzon(int numero)
{
  bool flagIndicador = false;
  Buzon anterior, actual;
  Monitor.Enter(this);

  anterior = actual = buzones;

  while (numero != actual.BuzonId && actual.Siguiente != null)
  {
    anterior = actual;
    actual = actual.Siguiente;
  }

  if (numero == actual.BuzonId
    && actual.PrimerProceso == null
    && actual.PrimerMensaje == null)
  {
    if (actual == anterior)
      buzones = buzones.Siguiente;
    else
      anterior.Siguiente = actual.Siguiente;

    flagIndicador = true;
  }
  Monitor.Exit(this);
  return flagIndicador;
}

public ArrayList LeerBuzones()
{
  ArrayList mLista = new ArrayList();
```

```
      Buzon actual = buzones;
      while(actual != null)
      {
        mLista.Add("Buzon " + actual.BuzonId);
        actual = actual.Siguiente;
      }
      return mLista;
    }

    public ArrayList LeerBuzonesElegibles()
    {
      ArrayList mLista = new ArrayList();
      Buzon actual = buzones;
      while (actual.BuzonId >= Niveles)
      {
        mLista.Insert(0,"Buzon " + actual.BuzonId);
        actual = actual.Siguiente;
      }

      return mLista;
    }

    public Buzon BuscarBuzon(int id)
    {
      Buzon actual = buzones;
      while (actual.Siguiente != null && actual.BuzonId != id)
      {
        actual = actual.Siguiente;
      }
      return actual;
    }
  }
}
```

///Paquete: Núcleo – núcleo del GASP

////Componente: GP1.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  public sealed partial  class GP : Listas
  {
    public GP()
    {
      inicializar();
      configurar();
    }

    public BaseTiempos BT
    {
      get
      {
        return bt;
      }
      set
```

```
      {
        bt = value;
      }
    }
  }
}
```

////Componente: GP2Enviar.cs

```
using System.Threading;

namespace GASP
{
  partial class GP
  {

    public void Enviar(Buzon buzon, Mensaje mensaje)
    {
      BCP receptor;
      semGEsspp.WaitOne();
      enGEsspp = true;

      if(verCambioModo)   //¿ ver ingreso a GPPE ?
        ingresarGPPE("Enviar");
/*=====================================================================
*/
      //bool esInterrupcion = EnEjecucion.Estado ==
ProcesoEstado.Expropiado;
      //if(!esInterrupcion)
      //  EnEjecucion.Estado = ProcesoEstado.Listo;
/*=====================================================================
*/

      if (buzon.HayProcesoEsperando())
      { // si hay receptor
        //puede que receptor tenga mas prioridad
        receptor = buzon.EliminarProcesoEsperando();

        //marcar mensaje consumido
        mensaje.Siguiente = mensaje;

        // mensaje se entrega a receptor
        receptor.Mensaje = mensaje;

        InsertarListo(receptor);

        if (verElProceso && EnEjecucion.Nombre == elProcesoNombre
          && receptor.Prioridad > EnEjecucion.Prioridad)
          mostrarEstado("Listo");
      }
      else
        buzon.InsertarMensajeEsperando(mensaje);

      planificar();
      despachar();
    }
  }
}
```

////Componente: GP3Recibir.cs

```
using System.Threading;
```

```
namespace GASP
{
  partial class GP
  {

    public void Recibir(Buzon buzon)
    {
      Mensaje mensaje;
      BCP proceso;
      semGEsspp.WaitOne();
      enGEsspp = true;

      if (verCambioModo) //¿ ver cambio de modo: (usu)<<>>GPPE ?
        ingresarGPPE("Recibir");
/*=======================================================*/
      //EnEjecucion.Estado = ProcesoEstado.Listo;
/*=======================================================*/
      proceso = EnEjecucion;

      if (buzon.HayMensajeEsperando())
      {
        // se extrae mensaje
        mensaje = buzon.EliminarMensajeEsperando();
        //mensaje señalizado, se entrega a proceso
        proceso.Mensaje = mensaje;
      }
      else
      { //no hay mensajes esperando
        //proceso se retira de cola "listos"
        EliminarListo();
        //proceso se en el buzon en cola de espera
        buzon.InsertarProcesoEsperando(proceso);

        if (verElProceso && proceso.Nombre == elProcesoNombre)
          mostrarEstado("Espera");
      }

      planificar();
      despachar();
    }
  }
}
```

////Componente: GP4planificar.cs

```
namespace GASP
{
  partial class GP
  {
    void planificar()
    {
      if (verConmutacion) //¿ ver origen de conmutacion ?
        mostrarDeProceso();

      if (verElProceso && EnEjecucion != Listos
        && Listos.Nombre == elProcesoNombre)
        mostrarEstado("EnEjecucion");

      //asigna el proceso listo con mayor prioridad
      //para ejecucion
      EnEjecucion = Listos;
```

```
      if (verConmutacion) //¿ ver destino de conmutacion ?
        mostrarAProceso();

      if (verCambioModo)    //¿ver salida de GPPE?
        salirGPPE();
    }
  }
}
```

////Componente: GP5despachar.cs

```
using System.Collections;

namespace GASP
{
  partial class GP
  {
    void despachar()
    {
      BCP proceso = EnEjecucion;

      /*asm: CARGAR REFERENCIAS HARD DE PILA DESDE BCP*/

      //mimica de establecer mascara del CIP para ejecución proceso
      BitArray mascaraTemp = new BitArray(Mascara);
      mascaraTemp.And(proceso.Mascara);
      mascaraTemp.Or(MascaraInicial);
      cip[dirES.cip01] = mascaraTemp;

      proceso.Estado = ProcesoEstado.EnEjecucion;
      enGEsspp = false;
      semGEsspp.Release();
    }
  }
}
```

////Componente: GP6Procesos.cs

```
using System;
using System.Threading;
using System.Collections;

namespace GASP
{
  partial class GP
  {

    public BCP CrearProceso(Thread proc, int prio, string nomb)
    {
      BCP proceso;
      semGEsspp.WaitOne();
      enGEsspp = true;

      proceso = InsertarProceso(proc, prio, nomb);

      if (verCambioModo) //¿ ver ingreso a GPPE ?
        ingresarGPPE(proceso.Nombre,"Crear");

      InsertarListo(proceso);
      enGEsspp = false;
      semGEsspp.Release();
```

```
      return proceso;
    }

    public void Fin()
    {
      semGEsspp.WaitOne();
      enGEsspp = true;

      if (verCambioModo)
        ingresarGPPE("Fin");

      //if (verElProceso && Listos.Nombre == elProcesoNombre)
      //  mostrarEstado("Finalizado");

      BCP proceso = EliminarListo();
      EliminarProceso(proceso);
      planificar();
      despachar();
    }
  }
}
```

////Componente: GP7Abortar.cs

```
using System.Threading;
namespace GASP
{
  partial class GP
  {
    internal void Abortar(string nomProceso)
    {
      semGEsspp.WaitOne();
      enGEsspp = true;
      BCP proceso = BuscarProceso(nomProceso);
      bool hayProceso = proceso != null;
      bool seEjecutaba = false;

      if(hayProceso)
        seEjecutaba = AbortarProceso(proceso);

      if (hayProceso && seEjecutaba)
      {
        planificar();
        despachar();
      }
      else
      {
        enGEsspp = false;
        semGEsspp.Release();
      }
    }

    public void Abortar(BCP proceso)
    {
      semGEsspp.WaitOne();
      enGEsspp = true;
      bool seEjecutaba = AbortarProceso(proceso);
      if (seEjecutaba)
      {
        planificar();
        despachar();
      }
```

```
      else
      {
        enGEsspp = false;
        semGEsspp.Release();
      }
    }
  }
}
```

////Componente: GP8Priorizar.cs

```
using System.Threading;
namespace GASP
{
  partial class GP
  {
    internal void Priorizar()
    {
      semGEsspp.WaitOne();
      enGEsspp = true;
      planificar();
      despachar();
    }
  }
}
```

////Componente: GP9Dormir.cs

```
using System.Threading;

namespace GASP
{
  partial class GP
  {
    public void Dormir(int uuTiempo)
    {
      semGEsspp.WaitOne();
      enGEsspp = true;
      if (uuTiempo > 0 && uuTiempo < 1001)
      {
        EliminarListo();
        InsertarDormido(uuTiempo);
      }
      planificar();
      despachar();
    }
  }
}
```

////Componente: GP10reloj.cs

```
namespace GASP
{
  partial class GP
  {
    private void reloj()
    {
      if (!enGEsspp && EnEjecucion.Estado ==
ProcesoEstado.EnEjecucion)
      {
        EnEjecucion.Reloj.Set();
      }
    }
```

```
  }
}
```

////Componente: GP11iniIRQx.cs

```
namespace GASP
{
  partial class GP
  {
    //nombres de interrupciones: IRQ0 ... IRQ7
    internal Mensaje IRQ0
    {
      get
      {
        return mensajeInterrupcion[(int)Nivel.Cero];
      }
    }

    //nombres usados para los buzones de mensajes de
    //interrupcion: BuzonIRQ0 ... BuzonIRQ7
    internal Buzon BuzonIRQ0
    {
      get
      {
        return buzonInterrupcion[(int)Nivel.Cero];
      }
    }
  }
}
```

////Componente: GP12interrupciones.cs

```
using System.Threading;

namespace GASP
{
  partial class GP
  {
    /* SERVIDOR DE INTERRUPCIONES */
    void servidorInterrups(Nivel nivelHard)
    {
      Monitor.Enter(this);

      if (verInterrupcion)  //¿ mostrar interrupcion ?
        mostrarInterrupcion(nivelHard);

      if (VerElProceso && EnEjecucion.Nombre == elProcesoNombre)  //¿
mostrar estado ?
        mostrarEstado("Expropiado");

      EnEjecucion.Estado = ProcesoEstado.Expropiado;
      Mensaje mensaje = mensajeInterrupcion[(int)nivelHard];
      Buzon buzon = buzonInterrupcion[(int)nivelHard];

      //finalizacion de pedido de interrupcion por dispositivo
      cip[dirES.cip00] = FinDeInt;

      //señalizacion de hilo que atiende interrupcion
      if (mensaje.Siguiente == mensaje)
      {//mensaje anterior fue consumido
        mensaje.Tipo = MensajeTipo.Interrupcion;
        Enviar(buzon, mensaje);
```

```
      }
      else
      {
        mensaje.Tipo = MensajeTipo.Perdido;
        Priorizar();
      }

      Monitor.Exit(this);
    }

    internal void HabilitarNivelInterrup(Nivel nivel)
    {
      Mascara.Set((int)nivel, true);
    }
  }
}
```

////Componente: GP13inivars.cs

```
using System.Text;
using System.Threading;

namespace GASP
{
  partial class GP
  {
    //semaforo binario que protege el sistema "gesspp"
    internal Semaphore semGEsspp;

    //indicador de modo: true = "sistema", false = "usuario"
    internal bool enGEsspp;

    //por herencia se inicializan mascara y listas

    //hardware
    private CIP cip; //Controlador de Interrucciones Programables

    //mensajes para señalización de interrucciones
    private Mensaje[] mensajeInterrupcion;

    //buzones para mensajes de interrupcion;
    private Buzon[] buzonInterrupcion;

    //referencias a objetos de gestión de interrupciones
    private Interrupcion HardIRQ0;

    //BASE DE TIEMPO
    //que define la frecuencia de reloj y uuTiempo
    private BaseTiempos bt;
  }
}
```

////Componente: GP14inicializar.cs

```
using System.Threading;

namespace GASP
{
  partial class GP
  {
    void inicializar()
    {
```

```
      //ingreso a proceso en el sistema
      semGEsspp = new Semaphore(0, 1);
      enGEsspp = true;

      cip = new CIP();//con interrups deshbilitadas

      //MASCARA DE INTERRUPCIONES DEL SISTEMA
      //inicializada al valor dado por "MascaraInicial"
      Mascara.Or(MascaraInicial);

      //MASCARA DE INTERRUPCIONES DEL CIP
      //establecido para que permita inicializar el sistema
      //cip[dirES.cip01].Xor(cip[dirES.cip01]);

      //definicion del numero de mensajes de interrupcion
      mensajeInterrupcion = new Mensaje[Niveles];

      //definicion del numero de buzones de interrupcion
      buzonInterrupcion = new Buzon[Niveles];

      //creación e inicializacion de mensajes y buzones
      //de interrupcion
      for (int nivel = 0; nivel < Niveles; nivel++)
      {
        mensajeInterrupcion[nivel] = new Mensaje();
        mensajeInterrupcion[nivel].Siguiente =
mensajeInterrupcion[nivel];
        mensajeInterrupcion[nivel].Tipo = MensajeTipo.Interrupcion;
        buzonInterrupcion[nivel] = InsertarBuzon();
      }

      //creacion de objetos que captan la interrupcion
      //y su conexion al conmutador a servicios de cada una
      HardIRQ0 = new Interrupcion(Nivel.Cero, cip.Mascara);
      HardIRQ0.Servidor += new
ManejadorInterrupcion(servidorInterrups);

      //creacion de la base de tiempor
      bt = new BaseTiempos();
      //Conexion del reloj del sistema a la base de tiempos
      bt.reloj = new GeneradorReloj(reloj);
      //Conexion del generador de interrupcion "HardIRQ0"
      //a la fuente de interrupcion implementada en la bt
      bt.IRQ0 = new GeneradorInterrupcion(HardIRQ0.Interrumpir);
      //señalar semaforo
      enGEsspp = false;
      semGEsspp.Release();
    }
  }
}
```

////Componente: GP15iniconfigurar.cs

```
using System.Threading;

namespace GASP
{
  partial class GP
  {
    void configurar()
    {
      //crear subProceso siempreListo
```

```
      pSiempreListo sp = new pSiempreListo(this);
      Thread pHacer = new Thread(new ThreadStart(sp.Hacer));
      pHacer.Name = "sListo";
      sp.bcpRef = CrearProceso(pHacer, 0, "sListo");


      //crear subProceso despertador
      pDespertador desp = new pDespertador(this);
      Thread pDespertar = new Thread(new ThreadStart(desp.Despertar));
      pDespertar.Name = "sDespertador";
      desp.bcpRef = CrearProceso(pDespertar, 255, "sDespertador");

      //poner a punto el inicio del sistema
      semGEsspp.WaitOne();
      enGEsspp = true;
      planificar();
      despachar();
      pDespertar.Start();
      pHacer.Start();
    }
  }
}
```

////Componente: GP16CrearBuzon.cs

```
using System.Threading;

namespace GASP
{
  partial class GP
  {
    public Buzon CrearBuzon()
    {
      semGEsspp.WaitOne();
      enGEsspp = true;
      Buzon buzon = InsertarBuzon();
      enGEsspp = false;
      semGEsspp.Release();
      return buzon;
    }
  }
}
```

////Componente: GP17DestruirBuzon.cs

```
using System.Threading;

namespace GASP
{
  partial class GP
  {
    public bool DestruirBuzon(int buzonNum)
    {
      bool flagBuzon;
      semGEsspp.WaitOne();
      enGEsspp = true;
      flagBuzon = EliminarBuzon(buzonNum);
      enGEsspp = false;
      semGEsspp.Release();
      return flagBuzon;
    }
  }
```

```
}
```

////Componente: GP18DisponerGEsspp.cs

```
using System.Threading;

namespace GASP
{
  partial class GP
  {
    public void DisponerGEsspp()
    {
      semGEsspp.WaitOne();
      enGEsspp = true;
      BT.Habilitado = false;
      DisponerProcesos();
      enGEsspp = false;
      semGEsspp.Release();
    }
  }
}
```

///Paquete: IGestor – interfaz del GASP

////Componente: IUBaseTiempos.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUBaseTiempos : Form
  {
    private BaseTiempos objBaseTiempos;
    ToolStripMenuItem cmdItem;
    public IUBaseTiempos(BaseTiempos bt, ToolStripMenuItem cmdIt)
    {
      InitializeComponent();
      cmdIt.Enabled = false;
      cmdItem = cmdIt;
      objBaseTiempos = bt;
      InicializarBT();
    }

    private void InicializarBT()
    {
      nudPeriodo.Value = objBaseTiempos.Periodo;
      nuduuTiempo.Value = objBaseTiempos.PeriodosXUT;
      presentarInterfase();
    }

    private void presentarInterfase()
    {
      if (objBaseTiempos.Habilitado)
      {
        nudPeriodo.Enabled = false;
        nuduuTiempo.Enabled = false;
```

```
        bnArrancar.Enabled = false;
        bnParar.Enabled = true;
      }
      else
      {
        nudPeriodo.Enabled = true;
        nuduuTiempo.Enabled = true;
        bnArrancar.Enabled = true;
        bnParar.Enabled = false;
      }
    }

    private void bnArrancar_Click(object sender, EventArgs e)
    {
      objBaseTiempos.Periodo = (int)nudPeriodo.Value;
      objBaseTiempos.PeriodosXUT = (int)nuduuTiempo.Value;
      objBaseTiempos.Habilitado = true;
      presentarInterfase();
    }

    private void bnParar_Click(object sender, EventArgs e)
    {
      objBaseTiempos.Habilitado = false;
      presentarInterfase();
    }

    private void IUBaseTiempos_FormClosing(object sender,
FormClosingEventArgs e)
    {
      cmdItem.Enabled = true;
    }
  }
}
```

////Componente: IUBuzones.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUBuzones : Form
  {
    GP gp;
    EventHandler manejadorParticular;
    ToolStripMenuItem cmdItem;
    public IUBuzones(GP sistema, ToolStripMenuItem cmdIt)
    {
      InitializeComponent();
      cmdIt.Enabled = false;
      cmdItem = cmdIt;
      gp = sistema;
      lbBuzones.DataSource = gp.LeerBuzones();
      lbBuzones.ClearSelected();
      display.Text = "Buzones 0:" + (gp.Niveles -1)+ " no son
eliminables...";
      lbBuzones.SelectedIndexChanged += manejadorParticular
```

```
      = new EventHandler(lbBuzones_SelectedIndexChanged);
    }

    private void bnAdicionar_Click(object sender, EventArgs e)
    {
      Buzon buzon = gp.CrearBuzon();
      lbBuzones.DataSource = gp.LeerBuzones();

      lbBuzones.SelectedIndexChanged -= manejadorParticular;
      lbBuzones.ClearSelected();
      lbBuzones.SelectedIndexChanged += manejadorParticular;

      display.Text = "Buzon " + lbBuzones.Items[0].ToString().Trim() +
" adicionado...";
      bnEliminar.Enabled = false;
    }

    private void lbBuzones_SelectedIndexChanged(object sender,
EventArgs e)
    {
      if (lbBuzones.SelectedIndex < (lbBuzones.Items.Count -
gp.Niveles))
      {
        bnEliminar.Enabled = true;
        display.Text = "Eliminar un Buzon puede dañar el programa
...";
      }
      else
      {
        bnEliminar.Enabled = false;
        lbBuzones.SelectedIndexChanged -= manejadorParticular;
        lbBuzones.ClearSelected();
        lbBuzones.SelectedIndexChanged += manejadorParticular;
        display.Text = "Buzones 0:" + (gp.Niveles - 1) + " no son
eliminables...";
      }
    }

    private void bnEliminar_Click(object sender, EventArgs e)
    {
      string cadena = lbBuzones.SelectedItem.ToString().Trim();
      cadena = cadena.Remove(0, "Buzon ".Length);
      int buzonNumero = int.Parse(cadena);
      bool flagIndicador = gp.DestruirBuzon(buzonNumero);
      if (flagIndicador)
      {
        lbBuzones.DataSource = gp.LeerBuzones();

        lbBuzones.SelectedIndexChanged -= manejadorParticular;
        lbBuzones.ClearSelected();
        lbBuzones.SelectedIndexChanged += manejadorParticular;
        bnEliminar.Enabled = false;

        display.Text = "Buzon '" + buzonNumero + "' eliminado...";
      }
      else
      {
        display.Text = "Buzon '" + buzonNumero + "' tiene 'cola' de
espera...";
      }
    }
```

```
    private void IUBuzones_FormClosing(object sender,
FormClosingEventArgs e)
    {
      cmdItem.Enabled = true;
    }
  }
}
```

////Componente: IUConsola.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUconsola : Form
  {
    GP gp;
    ToolStripMenuItem cmdItem;
    System.EventHandler chbProcesoHandler;
    event ManejadorConsola manejadorConsola;
    public IUconsola(GP sistema, ToolStripMenuItem cmdIt)
    {
      InitializeComponent();
      cmdIt.Enabled = false;
      cmdItem = cmdIt;
      gp = sistema;
      txtProceso.Text = gp.ElProcesoNombre;
      gp.Remitir += manejadorConsola
      = new ManejadorConsola(pMonitor_Remitir);
      this.chbProceso.CheckedChanged += chbProcesoHandler
      = new System.EventHandler(this.chbProceso_CheckedChanged);
    }

    private void chbProceso_CheckedChanged(object sender, EventArgs e)
    {
      if (chbProceso.Checked)
      {
        if (txtProceso.Text != "")
        {
          gp.ElProcesoNombre = txtProceso.Text;
          gp.VerElProceso = true;
          txtProceso.ReadOnly = true;
        }
        else
        {
          this.chbProceso.CheckedChanged -= chbProcesoHandler;
          display.AppendText("Ingrese nombre de sub proceso\n");
          chbProceso.Checked = false;
          this.chbProceso.CheckedChanged += chbProcesoHandler;
        }
      }
      else
      {
        gp.VerElProceso = false;
        txtProceso.ReadOnly = false;
```

```
      }
    }

    private void chbModo_CheckedChanged(object sender, EventArgs e)
    {
      if (chbModo.Checked)
        gp.VerCambioModo = true;
      else
        gp.VerCambioModo = false;
    }

    private void chbConmutacion_CheckedChanged(object sender,
EventArgs e)
    {
      if (chbConmutacion.Checked)
        gp.VerConmutacion = true;
      else
        gp.VerConmutacion = false;
    }

    private void chbInterrupcion_CheckedChanged(object sender,
EventArgs e)
    {
      if (chbInterrupcion.Checked)
        gp.VerInterrupcion = true;
      else
        gp.VerInterrupcion = false;
    }

    private void pMonitor_Remitir(string texto)
    {
      BeginInvoke(new ManejadorConsola(displayar), new Object[] {
texto });
    }

    private void displayar(string s)
    {
      display.AppendText(s);
      display.ScrollToCaret();
    }

    private void IUconsola_FormClosing(object sender,
FormClosingEventArgs e)
    {
      gp.VerElProceso = false;
      gp.VerCambioModo = false;
      gp.VerConmutacion = false;
      gp.VerInterrupcion = false;
      gp.Remitir -= manejadorConsola;
      cmdItem.Enabled = true;
    }

    private void bnBorrar_Click(object sender, EventArgs e)
    {
      display.Text = "";
    }
  }
}
```

////Componente: IUMensajes.cs

```
using System;
```

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace GASP
{
  public partial class IUMensajes : Form
  {
    GP gp;
    Buzon buzon;
    MensajeTipo tipo;
    Mensaje mensaje;
    ToolStripMenuItem cmdItem;
    bool reEvento = false;
    delAlBuzon revAlBuzonM;
    delAlBuzon revAlBuzonP;
    delDelBuzon revDelBuzonM;
    delDelBuzon revDelBuzonP;
    delForzarBuzon revForzarBuzonM;
    delForzarBuzon revForzarBuzonP;
    public IUMensajes(GP sistema, ToolStripMenuItem cmdIt)
    {
      InitializeComponent();
      cmdIt.Enabled = false;
      cmdItem = cmdIt;
      gp = sistema;

      cboTipo.DataSource = Enum.GetNames(typeof(MensajeTipo));

      revAlBuzonM = new delAlBuzon(buzon_evAlBuzonM);
      revAlBuzonP = new delAlBuzon(buzon_evAlBuzonP);
      revDelBuzonM = new delDelBuzon(buzon_evDelBuzonM);
      revDelBuzonP = new delDelBuzon(buzon_evDelBuzonP);
      revForzarBuzonM = new delForzarBuzon(buzon_evForzarBuzonM);
      revForzarBuzonP = new delForzarBuzon(buzon_evForzarBuzonP);
      cboBuzon.DataSource = gp.LeerBuzones();
    }

    private void IUMensajes_FormClosing(object sender,
FormClosingEventArgs e)
    {
        buzon.evAlBuzonM -= revAlBuzonM;
        buzon.evAlBuzonP -= revAlBuzonP;
        buzon.evDelBuzonM -= revDelBuzonM;
        buzon.evDelBuzonP -= revDelBuzonP;
        buzon.evForzarBuzonM -= revForzarBuzonM;
        buzon.evForzarBuzonP -= revForzarBuzonP;
        cmdItem.Enabled = true;
    }

    private void cboBuzon_SelectedIndexChanged(object sender,
EventArgs e)
    {
      string s = ((string)cboBuzon.SelectedItem).Trim();
      s = s.Remove(0,"Buzon ".Length);
      int buzonId = int.Parse(s);
```

```
      if (reEvento)
      {
        buzon.evAlBuzonM -= revAlBuzonM;
        buzon.evAlBuzonP -= revAlBuzonP;
        buzon.evDelBuzonM -= revDelBuzonM;
        buzon.evDelBuzonP -= revDelBuzonP;
        buzon.evForzarBuzonM -= revForzarBuzonM;
        buzon.evForzarBuzonP -= revForzarBuzonP;
      }
      reEvento = true;

      buzon = gp.BuscarBuzon(buzonId);

      buzon.evAlBuzonM += revAlBuzonM;
      buzon.evAlBuzonP += revAlBuzonP;
      buzon.evDelBuzonM += revDelBuzonM;
      buzon.evDelBuzonP += revDelBuzonP;
      buzon.evForzarBuzonM += revForzarBuzonM;
      buzon.evForzarBuzonP += revForzarBuzonP;

      lbMensajes.Items.Clear();
      lbMensajes.Items.AddRange(buzon.LeerMensajes().ToArray());
      if (lbMensajes.Items.Count > 0)
        bnRemover.Enabled = true;
      else
        bnRemover.Enabled = false;

      lbProcesos.Items.Clear();
      lbProcesos.Items.AddRange(buzon.LeerProcesos().ToArray());
    }

    private void cboTipo_SelectedIndexChanged(object sender, EventArgs
e)
    {
      string s = (string)cboTipo.SelectedItem;
      switch (s)
      {
        case "Normal":
          tipo = MensajeTipo.Normal;
          break;
        case "Interrupcion":
          tipo = MensajeTipo.Interrupcion;
          break;
        default:
          tipo = MensajeTipo.Perdido;
          break;
      }
    }

    private void bnAdicionar_Click(object sender, EventArgs e)
    {
      bnAdicionar.Enabled = false;
      bnRemover.Enabled = false;
      ControlBox = false;
      mensaje = new Mensaje();
      mensaje.Tipo = tipo;
      if (txtCuerpo.Text.Length > 0)
        mensaje.Cuerpo = txtCuerpo.Text;

      pMjeIngresar objIn = new pMjeIngresar(gp, mensaje, 1);
      Thread spEjecutar = new Thread(new ThreadStart(objIn.Ejecutar));
```

```
      string nombre = "p" + BCP.Todos + "MJEin";
      objIn.bcpRef = gp.CrearProceso(spEjecutar, 225, nombre);
      objIn.buzonRef = buzon;
      objIn.evFin += new IndicaDatoInt(this.objIn_evFin);
      spEjecutar.Name = nombre;
      spEjecutar.Start();
      gp.Priorizar();
    }

    private void bnRemover_Click(object sender, EventArgs e)
    {
      bnRemover.Enabled = false;
      bnAdicionar.Enabled = false;
      ControlBox = false;
      pMjeRetirar objOut = new pMjeRetirar(gp, 1);
      Thread spEjecutar = new Thread(new
ThreadStart(objOut.Ejecutar));
      string nombre = "p" + BCP.Todos + "MJEout";
      objOut.bcpRef = gp.CrearProceso(spEjecutar, 225, nombre);
      objOut.buzonRef = buzon;
      objOut.evFin += new IndicaDatoInt(this.objOut_evFin);
      spEjecutar.Name = nombre;
      spEjecutar.Start();
      gp.Priorizar();
    }

    private void buzon_evDelBuzonP()
    {
      BeginInvoke(new delDelBuzon(eliminarP));
    }

    private void buzon_evAlBuzonP(string nomProceso)
    {
      BeginInvoke(new delAlBuzon(insertarP), new Object[] { nomProceso
});
    }

    private void buzon_evDelBuzonM()
    {
      BeginInvoke(new delDelBuzon(eliminarM));
    }

    private void buzon_evAlBuzonM(string mje)
    {
      BeginInvoke(new delAlBuzon(insertarM), new Object[] { mje });
    }

    private void buzon_evForzarBuzonM(string mje)
    {
      BeginInvoke(new delAlBuzon(forzarM), new Object[] { mje });
    }

    private void buzon_evForzarBuzonP(string nomProceso)
    {
      BeginInvoke(new delAlBuzon(forzarP), new Object[] { nomProceso
});
    }

    private void eliminarP()
    {
      if(lbProcesos.Items.Count > 0)
```

```
        lbProcesos.Items.RemoveAt(0);
    }

    private void insertarP(string s)
    {
      lbProcesos.Items.Add(s);
    }

    private void forzarP(string s)
    {
      lbProcesos.SelectedItem = s;
      int indice = lbProcesos.SelectedIndex;
      if (indice > -1)
        lbProcesos.Items.RemoveAt(indice);
    }

    private void eliminarM()
    {
      if(lbMensajes.Items.Count > 0)
        lbMensajes.Items.RemoveAt(0);
    }

    private void insertarM(string m)
    {
      lbMensajes.Items.Add(m);
    }

    private void forzarM(string m)
    {
      lbMensajes.SelectedItem = m;
      int indice = lbMensajes.SelectedIndex;
      if (indice > -1)
        lbMensajes.Items.RemoveAt(indice);
    }

    private void objIn_evFin(int numMjes)
    {
      BeginInvoke(new IndicaDatoInt(habilitarAdicionar), new Object[]
{ numMjes });
    }

    private void objOut_evFin(int numMjes)
    {
      BeginInvoke(new IndicaDatoInt(manejarRemover), new Object[] {
numMjes });
    }

    private void habilitarAdicionar(int numero)
    {
      bnAdicionar.Enabled = true;
      bnRemover.Enabled = true;
      ControlBox = true;
    }

    private void manejarRemover(int numero)
    {
      txtCuerpo.Text = "Se removio " + numero + " mensaje";
      bnAdicionar.Enabled = true;
      if (lbMensajes.Items.Count > 0)
        bnRemover.Enabled = true;
      ControlBox = true;
```

```
    }
  }
}
```

////Componente: IUProcesos.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUprocesos : Form
  {
    GP gp;
    bool eliminado;
    ToolStripMenuItem cmdItem;
    delProcesoCreado revProCreado;
    delProcesoEliminado revProEliminado;
    delProcesoAbortado revProAbortado;

    public IUprocesos(GP sistema, ToolStripMenuItem cmdIt)
    {
      InitializeComponent();
      cmdIt.Enabled = false;
      cmdItem = cmdIt;
      gp = sistema;
      gp.evProCreado += revProCreado
        = new delProcesoCreado(gp_evProCreado);
      gp.evProEliminado += revProEliminado
        = new delProcesoEliminado(gp_evProEliminado);
      gp.evProAbortado += revProAbortado
        = new delProcesoAbortado(gp_evProAbortado);

      dgvProcesos.ColumnCount = 3;
      dgvProcesos.Columns[0].Name = "subProceso";
      dgvProcesos.Columns[1].Name = "Prioridad";
      dgvProcesos.Columns[2].Name = "Tipo";
      IEnumerable  pp = gp.ListarBCPs();
      foreach(object p in pp)
      {
        string[] ss = p.ToString().Split(new Char[] {','});
        dgvProcesos.Rows.Add(ss);
      }

      for (int i = 0; i < dgvProcesos.ColumnCount; i++)
        dgvProcesos.Columns[i].Width = (ClientSize.Width -2) /
dgvProcesos.ColumnCount;
    }

    private void bnAbortar_Click(object sender, EventArgs e)
    {
      if (dgvProcesos.SelectedRows.Count > 0)
      {
        int indice = dgvProcesos.SelectedRows[0].Index;
        string nombre =
(string)dgvProcesos.Rows[indice].Cells[0].Value;
        if (nombre != "sDespertador" && nombre != "sListo")
        {
```

```
            gp.Abortar(nombre);
            dgvProcesos.Rows.RemoveAt(indice);
            display.Text = "sub Proceso '" + nombre + "' abortado";
          }
          else
            display.Text = "'sDespertador' y 'sListo' no son
eliminables";
        }
        else
          display.Text = "elija proceso a eliminar...";
      }

      private void IUprocesos_FormClosing(object sender,
FormClosingEventArgs e)
      {
        gp.evProCreado -= revProCreado;
        gp.evProEliminado -= revProEliminado;
        gp.evProAbortado -= revProAbortado;
        cmdItem.Enabled = true;
      }

      private void dgvProcesos_SelectionChanged(object sender, EventArgs
e)
      {
        display.Text = "";
      }

      private void gp_evProCreado(string[] datos)
      {
        BeginInvoke(new delProcesoCreado(adicionarProceso), new Object[]
{ datos });
      }

      private void gp_evProEliminado(string nomProceso)
      {
        eliminado = true;
        BeginInvoke(new delProcesoEliminado(eliminarProceso), new
Object[] { nomProceso});
      }

      private void gp_evProAbortado(string nomProceso)
      {
        eliminado = false;
        BeginInvoke(new delProcesoAbortado(eliminarProceso), new
Object[] { nomProceso});
      }

      private void adicionarProceso(string[] dd)
      {
        dgvProcesos.Rows.Add(dd);
        display.Text = "";
      }

      private void eliminarProceso(string nom)
      {
        int fila = 0;
        while (fila < dgvProcesos.Rows.Count
          && nom != dgvProcesos.Rows[fila].Cells[0].Value.ToString())
          fila++;
        if (fila < dgvProcesos.Rows.Count)
        {
```

```
        dgvProcesos.Rows.RemoveAt(fila);
      }

      if (eliminado)
        display.Text = nom + ": finalizado...";
      else
        display.Text = nom + ": abortado...";
    }
  }
}
```

////Componente: IUBuzon.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUBuzon : Form
  {
    Buzon buzon;
    delAlBuzon revAlBuzonM;
    delAlBuzon revAlBuzonP;
    delDelBuzon revDelBuzonM;
    delDelBuzon revDelBuzonP;
    delForzarBuzon revForzarP;
    delForzarBuzon revForzarM;
    public IUBuzon(Buzon buzonRef)
    {
      InitializeComponent();
      buzon = buzonRef;
      buzon.evAlBuzonM += revAlBuzonM
        = new delAlBuzon(buzon_evAlBuzonM);
      buzon.evAlBuzonP += revAlBuzonP
        = new delAlBuzon(buzon_evAlBuzonP);
      buzon.evDelBuzonM += revDelBuzonM
        = new delDelBuzon(buzon_evDelBuzonM);
      buzon.evDelBuzonP += revDelBuzonP
        = new delDelBuzon(buzon_evDelBuzonP);
      buzon.evForzarBuzonM += revForzarM
        = new delForzarBuzon(buzon_evForzarBuzonM);
      buzon.evForzarBuzonP += revForzarP
        = new delForzarBuzon(buzon_evForzarBuzonP);

      Text += " " + buzon.BuzonId;
      buzon.Vista = this;
      lbMensajes.Items.AddRange(buzon.LeerMensajes().ToArray());
      lbProcesos.Items.AddRange(buzon.LeerProcesos().ToArray());
    }

    private void buzon_evDelBuzonP()
    {
      BeginInvoke(new delDelBuzon(eliminarP));
    }

    private void buzon_evAlBuzonP(string nomProceso)
    {
```

```
      BeginInvoke(new delAlBuzon(insertarP), new Object[] { nomProceso
});
    }

    private void buzon_evDelBuzonM()
    {
      BeginInvoke(new delDelBuzon(eliminarM));
    }

    private void buzon_evAlBuzonM(string mje)
    {
      BeginInvoke(new delAlBuzon(insertarM), new Object[] { mje });
    }

    private void buzon_evForzarBuzonM(string mje)
    {
      BeginInvoke(new delAlBuzon(forzarM), new Object[] { mje });
    }

    private void buzon_evForzarBuzonP(string nomProceso)
    {
      BeginInvoke(new delAlBuzon(forzarP), new Object[] { nomProceso
});
    }

    private void eliminarP()
    {
      lbProcesos.Items.RemoveAt(0);
    }

    private void insertarP(string s)
    {
      lbProcesos.Items.Add(s);
    }

    private void forzarP(string s)
    {
      lbProcesos.SelectedItem = s;
      int indice = lbProcesos.SelectedIndex;
      if (indice > -1)
        lbProcesos.Items.RemoveAt(indice);
    }

    private void eliminarM()
    {
      if(lbMensajes.Items.Count > 0)
        lbMensajes.Items.RemoveAt(0);
    }

    private void insertarM(string m)
    {
      lbMensajes.Items.Add(m);
    }

    private void forzarM(string m)
    {
      lbMensajes.SelectedItem = m;
      int indice = lbMensajes.SelectedIndex;
      if (indice > -1)
        lbMensajes.Items.RemoveAt(indice);
    }
```

```
    private void IUBuzon_FormClosing(object sender,
FormClosingEventArgs e)
    {
      buzon.evAlBuzonM -= revAlBuzonM;
      buzon.evAlBuzonP -= revAlBuzonP;
      buzon.evDelBuzonM -= revDelBuzonM;
      buzon.evDelBuzonP -= revDelBuzonP;
      buzon.evForzarBuzonM -= revForzarM;
      buzon.evForzarBuzonP -= revForzarP;
      buzon.Vista = null;
    }
  }
}
```

////Componente: IUMemoria.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUMemoria : Form
  {
    MemCompartida mC;
    string ultimaOper = " ", ultimoSubP = " ";
    VerBufferUno revMostrar;
    public IUMemoria(object mostrado)
    {
      InitializeComponent();
      mC = (MemCompartida)mostrado;
      mC.Vista = this;
      lblMemoria.Text = mC.Valor.ToString();
      mC.Mostrar += revMostrar = new VerBufferUno(memoria_Mostrar);
    }

    private void memoria_Mostrar(string oper, string nomSubP, int
dato)
    {
      BeginInvoke(new VerBufferUno(mostrarMC), new Object[] { oper,
nomSubP, dato });
    }

    private void mostrarMC(string operacion, string nomSubP, int dato)
    {
      string sdato = dato.ToString();
      if (operacion == "E") lblMemoria.Text = sdato;

      sdato += operacion == "E" ? "=>" : "<=";

      if (ultimaOper == "L" && operacion == "E" && ultimoSubP !=
nomSubP)
        sdato += "?";

      lbAccesos.Items.Add(nomSubP + ": " + sdato);

      ultimaOper = operacion;
```

```
      ultimoSubP = nomSubP;
    }

    private void bnBorrar_Click(object sender, EventArgs e)
    {
      lbAccesos.Items.Clear();
    }

    private void IUMemoria_FormClosing(object sender,
FormClosingEventArgs e)
    {
      mC.Mostrar -= revMostrar;
      mC.Vista = null;
    }
  }
}
```

////Componente: IUBufferUno.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUBufferUno : Form
  {
    BufferUno buffer;
    string ultimaOper = " ";
    VerBufferUno revMostrar;
    public IUBufferUno(object mostrado)
    {
      InitializeComponent();

      buffer = (BufferUno)mostrado;
      lblMemoria.Text = buffer.Valor.ToString();
      buffer.Mostrar += revMostrar = new VerBufferUno(buffer_Mostrar);
      buffer.Vista = this;
    }

    private void buffer_Mostrar(string oper, string nomSubP, int dato)
    {
      BeginInvoke(new VerBufferUno(mostrar), new Object[] {oper,
nomSubP, dato });
    }

    private void mostrar(string operacion, string nomSubP, int dato)
    {
      string sdato = dato.ToString();
      if(operacion == "P")  lblMemoria.Text = sdato;

      if(operacion == ultimaOper)
        sdato += operacion == "P" ? "!" : "¡";

      if (operacion == "P")
      {
        lbIngreso.Items.Insert(0, nomSubP + ": " + sdato);
      }
```

```
      else
        lbSalida.Items.Insert(0, nomSubP + ": " + sdato);

      ultimaOper = operacion;
    }

    private void IUBufferSingular_FormClosing(object sender,
FormClosingEventArgs e)
    {
      buffer.Mostrar -= revMostrar;
      buffer.Vista = null;
    }

    private void bnBorrarIngreso_Click(object sender, EventArgs e)
    {
      lbIngreso.Items.Clear();
    }

    private void bnBorrarSalida_Click(object sender, EventArgs e)
    {
      lbSalida.Items.Clear();
    }
  }
}
```

////Componente: IUBufferIli.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUBufferIli : Form
  {
    BufferIli buffer;
    VerBuffer revMostrar;
    public IUBufferIli(object mostrado)
    {
      InitializeComponent();

      buffer = (BufferIli)mostrado;

      lbMemoria.Items.AddRange(buffer.ValorActual);

      lblElementos.Text = "Disponibles: " + buffer.Valor;

      buffer.Mostrar += revMostrar = new VerBuffer(buffer_Mostrar);
      buffer.Vista = this;
    }

    private void buffer_Mostrar(string oper, string nomSubP, int dato,
int i, int elementos)
    {
      BeginInvoke(new VerBuffer(mostrar), new Object[] { oper,
nomSubP, dato, i, elementos });
    }
```

```
    private void mostrar(string operacion, string nomSubP, int dato,
int i, int elos)
    {
      if (operacion == "P") //escribir en buffer
      {
        lbIngreso.Items.Insert(0, nomSubP + "[" + i + "]: " + dato);
        lbMemoria.Items.Add(dato);
        lblElementos.Text = "Disponibles: " + elos;
      }
      else if (operacion == "C") // leer buffer
      {
        lbMemoria.Items.RemoveAt(0);
        lbSalida.Items.Insert(0, nomSubP + "[" + i + "]: " + dato);
        lblElementos.Text = "Disponibles: " + elos;
      }
      else //inicializar buffer
      {
        lbMemoria.Items.Clear();
        lbMemoria.Items.AddRange(buffer.ValorActual);
        lblElementos.Text = "Disponibles: " + elos;
      }
    }

    private void bnBorrarIngreso_Click(object sender, EventArgs e)
    {
      lbIngreso.Items.Clear();
    }

    private void bnBorrarSalida_Click(object sender, EventArgs e)
    {
      lbSalida.Items.Clear();
    }

    private void IUBufferIli_FormClosing(object sender,
FormClosingEventArgs e)
    {
      buffer.Mostrar -= revMostrar;
      buffer.Vista = null;
    }
  }
}
```

////Componente: IUBufferX.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace GASP
{
  public partial class IUBufferX : Form
  {
    BufferLim buffer;
    int longBuffer;
    VerBufferX revMostrar;
    public IUBufferX(object mostrado)
    {
      InitializeComponent();
```

```
      buffer = (BufferLim)mostrado;
      longBuffer = buffer.Valor;
      lbMemoria.Items.AddRange(buffer.ValorActual);

      lblElementos.Text = "Estado: 0/" + longBuffer;

      buffer.Mostrar += revMostrar = new VerBufferX(buffer_Mostrar);
      buffer.Vista = this;
    }

    private void buffer_Mostrar(string oper, string nomSubP, int dato,
int i, int elementos)
    {
      BeginInvoke(new VerBufferX(mostrar), new Object[] { oper,
nomSubP, dato, i, elementos });
    }

    private void mostrar(string operacion, string nomSubP, int dato,
int i, int elos)
    {
      if(operacion == "P") //escribir en buffer
      {
          lbIngreso.Items.Insert(0, nomSubP + "[" + i + "]: " + dato);
          lbMemoria.Items.RemoveAt(i);
          lbMemoria.Items.Insert(i, dato);
          lblElementos.Text = "Estado: " + elos + "/" + longBuffer;
      }
      else if(operacion == "C") // leer buffer
      {
          lbSalida.Items.Insert(0,nomSubP + "[" + i + "]: " + dato);
          lblElementos.Text = "Estado: " + elos + "/" + longBuffer;
      }
      else //inicializar buffer
      {
          lbMemoria.Items.Clear();
          lbMemoria.Items.AddRange(buffer.ValorActual);
          longBuffer = elos;
          lblElementos.Text = "Estado: 0/" + elos;
      }
    }

    private void IUBufferX_FormClosing(object sender,
FormClosingEventArgs e)
    {
      buffer.Mostrar -= revMostrar;
      buffer.Vista = null;
    }

    private void bnBorrarIngreso_Click(object sender, EventArgs e)
    {
      lbIngreso.Items.Clear();
    }

    private void bnBorrarSalida_Click(object sender, EventArgs e)
    {
      lbSalida.Items.Clear();
    }
  }
}
```

///Paquete: Tareas Específicas – subprocesos de soporte

////Componente: pDespertador.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  class pDespertador : Tarea
  {
    internal pDespertador(GP gpp) : base(gpp)
    {}

    internal void Despertar()
    {
      BCP proceso;
      gp.HabilitarNivelInterrup(Nivel.Cero);
      reloj.WaitOne();

      while (true)
      {
        //Console.Write(bcPropio.Nombre + ": Recibir => ");
        Recibir(gp.BuzonIRQ0);
        gp.semGEsspp.WaitOne();
        gp.enGEsspp = true;
        if (gp.Dormidos != null)
        {
          //existen procesos dormidos, el sueño del primer
          //dormido disminuye en una unidad de tiempo
          gp.Dormidos.DormirCuota--;

          //se despiertan todos los procesos que terminaron del dormir
          // y pasas al estado "listo"
          while (gp.Dormidos != null && gp.Dormidos.DormirCuota == 0)
          {
            proceso = gp.EliminarDormido();
            gp.InsertarListo(proceso);
          }
        }
        gp.enGEsspp = false;
        gp.semGEsspp.Release();
      }
    }
  }
}
```

////Componente: pSiempreListo.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  class pSiempreListo : Tarea
  {
    internal pSiempreListo(GP gpp) : base(gpp)
    {}
```

```
    public void Hacer()
    {
      int n = 0;

      while (true)
      {
        reloj.WaitOne();
        n = ++n % 1000;
      }
    }
  }
}
```

////Componente: pMjeIngresar.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  public class pMjeIngresar : Tarea
  {
    internal pMjeIngresar(GP gpp, Mensaje mje, int n)
      : base(gpp)
    {
      mensaje = mje;
      numero = n;
    }

    Mensaje mensaje;
    int numero;

    public event IndicaDatoInt evFin;

    public override void Ejecutar()
    {
      Mensaje mje;
      int n = 0;
      reloj.WaitOne();

      while (n < numero)
      {
        mje = new Mensaje(mensaje);
        Enviar(buzon, mje);
        n++;
        reloj.WaitOne();
      }

      if (evFin != null) evFin(n);

      gp.Fin();
    }
  }
}
```

////Componente: pMensajeRetirar.cs

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
using System.Threading;

namespace GASP
{
  public class pMjeRetirar : Tarea
  {
    internal pMjeRetirar(GP gpp, int n)
      : base(gpp)
    {
      numero = n;
    }

    int numero;

    public event IndicaDatoInt evFin;

    public override void Ejecutar()
    {
      int n = 0;
      reloj.WaitOne();

      while (n < numero)
      {
        Recibir(buzon);
        n++;
        reloj.WaitOne();
      }

      if (evFin != null) evFin(n);
      gp.Fin();
    }
  }
}
```

////Componente: Tarea.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace GASP
{
  public delegate void IndicaDatoString(string s);
  public delegate void IndicaDatoInt(int n);
  public delegate void IndicaDatoObject(object o);
  public delegate void IndicaCodigo(string s);
  public delegate void IndicaSignal();

  public class Tarea
  {
    protected GP gp;//nucleo de gestor de subprocesos
    protected BCP bcp;//descriptor del subproceso
    protected Buzon buzon;//buzones de mensajes
    protected Buzon buzonDos;
    protected Buzon buzonTres;
    protected AutoResetEvent reloj;//reloj de ejecucion

    public Tarea(GP gpp)
    {
      gp = gpp;
```

```
    }

    public BCP bcpRef
    {
      set
      {
        bcp = value;
        reloj = bcp.Reloj;
      }
    }

    public Buzon buzonRef
    {
      set
      {
        buzon = value;
      }
    }

    public Buzon buzonDosRef
    {
      set
      {
        buzonDos = value;
      }
    }

    public Buzon buzonTresRef
    {
      set
      {
        buzonTres = value;
      }
    }

    protected void Enviar(Buzon buzon, Mensaje mensaje)
    {
      gp.Enviar(buzon, mensaje);
      reloj.WaitOne();
    }

    protected Mensaje Recibir(Buzon buzon)
    {
      gp.Recibir(buzon);
      reloj.WaitOne();
      return bcp.Mensaje;
    }

    protected void Dormir(int uuTiempo)
    {
      gp.Dormir(uuTiempo);
      reloj.WaitOne();
    }

    public virtual void Ejecutar(){}
  }
}
```

////Componente: Via.cs

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
using System.Windows.Forms;
using System.Threading;

namespace GASP
{
  public class Via
  {
    public string clase;
    public Thread sp;
    public string nombre;
    public int prioridad;
    public object objCompartido;
    public Buzon buzon;
    public Buzon buzonDos;
    public Buzon buzonTres;
    public bool origen;

    public Via()
    {}

    public Via(Via o)
    {
    clase         = o.clase;
    sp            = o.sp;
    nombre        = o.nombre;
    prioridad     = o.prioridad;
    objCompartido = o.objCompartido;
    buzon         = o.buzon;
    buzonDos      = o.buzonDos;
    buzonTres     = o.buzonTres;
    origen        = o.origen;
    }
  }
}
```

B. Fuentes de IGASP en C#.

//UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
//FACULTAD DE CIENCIAS MATEMATICAS
//UNIDAD DE POST-GRADO
//MAESTRIA EN COMPUTACION E INFORMATICA
//TESIS DE GRADO: Ing. Francisco S. Aguilar V.
//FECHA: 2011
//**S**ubsistema IGASP – Interfaz de Integración del GASP y los subprocesos
// gestionados

////Componente: Programa.cs

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace IGASP
{
    static class Programa
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new IGI());
        }
    }
}
```

////Componente: IGI.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using GASP;
using subProcesos;

namespace IGASP
{

  public partial class IGI : Form
  {
    //GASP
    GP gp;
    //objeto para captura y transferencia de datos a subprocesos
    Via via;
    //objetos, compartidos
    MemCompartida mC;
    BufferUno bU;
    BufferLim bL;
    BufferIli bI;
```

```
    //ARREGLOS PARA PARAMETROS PREDEFINIDOS
    //para subprocesos independientes
    string[][] ssppCom = new string[2][];
    //para subprocesos colaborativos
    string[][] ssppCol = new string[3][];
    string[][] ssppColPares = new string[4][];
    //para subprocesos productores/consumidores
    string[][] ssppPPyCC = new string [4][];//un bufer
    string[][] ssppPPyCCx = new string[2][];//buferes limitados
    string[][] ssppPPyCCi = new string[2][];//buferes no limitados

    private void configuracionInicial()
    {
      //crear GASP
      gp = new GP();

      //crear objeto de captura de parametros
      via = new Via();

      //CREAR PARAMETROS PREDEFINIDOS para subprocesos organizados en
menus:
      //Menu: Competitivos
                                  //Clase          Busones Buffer Memoria
      ssppCom[0] = new string[]{"Sumatoria",     "0",    "0",
"0"};
      ssppCom[1] = new string[]{"Factorial",     "0",    "0",
"0"};
      //Menu Colaborativos, Item: UnTipo
      ssppCol[0] = new string[]{"Acumulador",    "0",    "0",
"1"};
      ssppCol[1] = new string[]{"AcumuladorP",   "0",    "0",
"1"};
      ssppCol[2] = new string[]{"AcumuladorExM", "1",    "0",
"1"};
      //Menu Colaborativos, Item: Complementados
      ssppColPares[0] = new string[]{"AcumuladorMSCo","2",  "0",
"1"};//primer
      ssppColPares[1] = new string[]{"AcumuladorMSig","2",  "0",
"1"};// par
      ssppColPares[2] = new string[]{"AcumuladorMTCo","2",  "0",
"0"};//segundo
      ssppColPares[3] = new string[]{"AcumuladorMTD", "2",  "0",
"0"};// par
      //Menu Casos, submenu Produc/Consum, item: Libres
      ssppPPyCC[0] = new string[]{"Productor",   "0",    "1",
"0"};//BufferUno
      ssppPPyCC[1] = new string[]{"Consumidor",  "0",    "1",
"0"};//BufferUno
      //Menu Casos, submenu Produc/Consum, item: Exclusion Mutua
      ssppPPyCC[2] = new string[]{"ProductorExM",   "2", "1",
"0"};//BufferUno
      ssppPPyCC[3] = new string[]{"ConsumidorExM",  "2", "1",
"0"};//BufferUno
      //Menu Casos, submenu Product/Consum, item: Señanizacion
      ssppPPyCCx[0]= new string[]{"ProductorLim",   "3", "1",
"0"};//BufferLim
      ssppPPyCCx[1]= new string[]{"ConsumidorLim",  "3", "1",
"0"};//BufferLim
      //Menu Casos, submenu Product/Consum, item: Comunicacion De
Datos
```

```
      ssppPPyCCi[0]= new string[]{"ProductorIli",   "2", "1",
"0"};//BufferIli
      ssppPPyCCi[1]= new string[]{"ConsumidorIli",  "2", "1",
"0"};//BufferIli
    }

    public IGI()
    {
      InitializeComponent();
      configuracionInicial();
    }

    private void cmdBaseTiempos_Click(object sender, EventArgs e)
    {
      IUBaseTiempos bt = new IUBaseTiempos(gp.BT, cmdBaseTiempos);
      bt.MdiParent = this;
      bt.Show();
    }

    private void cmdBuzones_Click(object sender, EventArgs e)
    {
      IUBuzones buzones = new IUBuzones(gp, cmdBuzones);
      buzones.MdiParent = this;
      buzones.Show();
    }

    private void cmdConsola_Click(object sender, EventArgs e)
    {
      IUconsola consola = new IUconsola(gp, cmdConsola);
      consola.MdiParent = this;
      consola.Show();
      consola = null;
    }

    private void cmdMensajes_Click(object sender, EventArgs e)
    {
      IUMensajes mjes = new IUMensajes(gp, cmdMensajes);
      mjes.MdiParent = this;
      mjes.Show();
    }

    private void cmdProcesos_Click(object sender, EventArgs e)
    {
      IUprocesos procesos = new IUprocesos(gp, cmdProcesos);
      procesos.MdiParent = this;
      procesos.Show();
    }

    private void cmdSalir_Click(object sender, EventArgs e)
    {
      gp.DisponerGEsspp();
      Application.Exit();
    }

    private void IGI_FormClosing(object sender, FormClosingEventArgs
e)
    {
      gp.DisponerGEsspp();
      Application.Exit();
    }
```

```
private void cmdSumat_Click(object sender, EventArgs e)
{
  Via oVia = new Via();
  IUpreIUsp pre = new IUpreIUsp(gp,oVia,ssppCom[0]);
  bool continuar = pre.ShowDialog(this) == DialogResult.OK;
  pre.Dispose();

  if (continuar)
  {
    IUSingular singular = new IUSingular(gp, oVia);
    singular.MdiParent = this;
    singular.Show();
  }
}

private void cmdFactor_Click(object sender, EventArgs e)
{
  Via oVia = new Via();
  IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppCom[1]);
  bool continuar = pre.ShowDialog(this) == DialogResult.OK;
  pre.Dispose();

  if (continuar)
  {
    IUSingular singular = new IUSingular(gp, oVia);
    singular.MdiParent = this;
    singular.Show();
  }
}

private void cmdAcumulador_Click(object sender, EventArgs e)
{
  Via oVia = new Via(via);
  if (mC == null) mC = new MemCompartida();
  oVia.objCompartido = mC;
  //oVia.tipoCompartido = TipoCompartido.Memoria;

  IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppCol[0]);
  bool continuar = pre.ShowDialog(this) == DialogResult.OK;
  pre.Dispose();

  if (continuar)
  {
    if (mC.Vista == null)
    {
      IUMemoria iuMem = new IUMemoria(mC);
      iuMem.MdiParent = this;
      iuMem.Show();
    }

    via = oVia;

    IUAcumulador acumulador = new IUAcumulador(gp, oVia);
    acumulador.MdiParent = this;
    acumulador.Show();
  }
}

private void AcumuladorE_Click(object sender, EventArgs e)
{
  Via oVia = new Via(via);
```

```
    if (mC == null) mC = new MemCompartida();
    oVia.objCompartido = mC;
    //oVia.tipoCompartido = TipoCompartido.Memoria;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppCol[1]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      if (mC.Vista == null)
      {
        IUMemoria iuMem = new IUMemoria(mC);
        iuMem.MdiParent = this;
        iuMem.Show();
      }

      via = oVia;

      IUAcumulador acumulador = new IUAcumulador(gp, oVia);
      acumulador.MdiParent = this;
      acumulador.Show();
    }
  }

  private void AcumuladorS_Click(object sender, EventArgs e)
  {
    Via oVia = new Via(via);
    if (mC == null) mC = new MemCompartida();
    oVia.objCompartido = mC;
    //oVia.tipoCompartido = TipoCompartido.Memoria;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppCol[2]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      if (mC.Vista == null)
      {
        IUMemoria iuMem = new IUMemoria(mC);
        iuMem.MdiParent = this;
        iuMem.Show();
      }

      if (oVia.buzon != null && oVia.buzon.Vista == null )
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzon);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }

      via = oVia;

      IUAcumulador acumulador = new IUAcumulador(gp, oVia);
      acumulador.MdiParent = this;
      acumulador.Show();
    }
  }

  private void cmdSigAcuSincCo_Click(object sender, EventArgs e)
```

```
{
  Via oVia = new Via(via);

  if (mC == null) mC = new MemCompartida();
  oVia.objCompartido = mC;
  //oVia.tipoCompartido = TipoCompartido.Memoria;

  IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppColPares[0]);
  bool continuar = pre.ShowDialog(this) == DialogResult.OK;
  pre.Dispose();

  if (continuar)
  {
    if (mC.Vista == null)
    {
      IUMemoria iuMem = new IUMemoria(mC);
      iuMem.MdiParent = this;
      iuMem.Show();
    }

    //buzon donde recibe mensajes
    if (oVia.buzonDos != null && oVia.buzonDos.Vista == null)
    {
      IUBuzon iuBuz = new IUBuzon(oVia.buzonDos);
      iuBuz.MdiParent = this;
      iuBuz.Show();
    }

    oVia.origen = true; //sp es fuente de mensajes
    via = oVia;

    IUAcumuladorPar acumuladorPar = new IUAcumuladorPar(gp, oVia);
    acumuladorPar.MdiParent = this;
    acumuladorPar.Show();
  }
}

private void cmdSigAcuSinc_Click(object sender, EventArgs e)
{
  Via oVia = new Via(via);

  if (mC == null) mC = new MemCompartida();
  oVia.objCompartido = mC;
  //oVia.tipoCompartido = TipoCompartido.Memoria;

  IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppColPares[1]);
  bool continuar = pre.ShowDialog(this) == DialogResult.OK;
  pre.Dispose();

  if (continuar)
  {
    if (mC.Vista == null)
    {
      IUMemoria iuMem = new IUMemoria(mC);
      iuMem.MdiParent = this;
      iuMem.Show();
    }

    //buzon donde recibe mensajes
    if (oVia.buzon != null && oVia.buzon.Vista == null)
    {
```

```
        IUBuzon iuBuz = new IUBuzon(oVia.buzon);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }

      oVia.origen = false; //sp es destino de mensajes
      via = oVia;

      IUAcumuladorPar acumuladorPar = new IUAcumuladorPar(gp, oVia);
      acumuladorPar.MdiParent = this;
      acumuladorPar.Show();
    }
  }

  private void cmdTDAcuSincCo_Click(object sender, EventArgs e)
  {
    Via oVia = new Via(via);
    oVia.objCompartido = null;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppColPares[2]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      //buzon donde recibe mensajes
      if (oVia.buzonDos != null && oVia.buzonDos.Vista == null)
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzonDos);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }

      oVia.origen = true; //sp es fuente de mensajes
      via = oVia;

      IUAcumuladorPar acumuladorPar = new IUAcumuladorPar(gp, oVia);
      acumuladorPar.MdiParent = this;
      acumuladorPar.Show();
    }
  }

  private void cmdTDAcuSinc_Click(object sender, EventArgs e)
  {
    Via oVia = new Via(via);
    oVia.objCompartido = null;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppColPares[3]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      //buzon donde recibe mensajes
      if (oVia.buzon != null && oVia.buzon.Vista == null)
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzon);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }
```

```
      oVia.origen = false; //sp es destino de mensajes
      via = oVia;

      IUAcumuladorPar acumuladorPar = new IUAcumuladorPar(gp, oVia);
      acumuladorPar.MdiParent = this;
      acumuladorPar.Show();
    }
  }

  private void cmdPPCCLibProd_Click(object sender, EventArgs e)
  {
    Via oVia = new Via(via);
    if (bU == null) bU = new BufferUno();
    oVia.objCompartido = bU;
    //oVia.tipoCompartido = TipoCompartido.BufferL;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppPPyCC[0]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      if (bU.Vista == null)
      {
        IUBufferUno iuBuf = new IUBufferUno(bU);
        iuBuf.MdiParent = this;
        iuBuf.Show();
      }

      oVia.origen = true; //sp es fuente de mensajes
      via = oVia;

      IUppYcc pYc = new IUppYcc(gp, oVia);
      pYc.MdiParent = this;
      pYc.Show();
    }
  }

  private void cmdPPCCLibCon_Click(object sender, EventArgs e)
  {
    Via oVia = new Via(via);
    if (bU == null) bU = new BufferUno();
    oVia.objCompartido = bU;
    //oVia.tipoCompartido = TipoCompartido.BufferL;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppPPyCC[1]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      if (bU.Vista == null)
      {
        IUBufferUno iuBuf = new IUBufferUno(bU);
        iuBuf.MdiParent = this;
        iuBuf.Show();
      }

      via = oVia;

      oVia.origen = false; //sp es destino de mensajes
```

```
    IUppYcc pYc = new IUppYcc(gp, oVia);
    pYc.MdiParent = this;
    pYc.Show();
  }
}

private void cmdPPCCExMProd_Click(object sender, EventArgs e)
{
  Via oVia = new Via(via);
  if (bU == null) bU = new BufferUno();
  oVia.objCompartido = bU;
  //oVia.tipoCompartido = TipoCompartido.BufferL;

  IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppPPyCC[2]);
  bool continuar = pre.ShowDialog(this) == DialogResult.OK;
  pre.Dispose();

  if (continuar)
  {
    if (bU.Vista == null)
    {
      IUBufferUno iuBuf = new IUBufferUno(bU);
      iuBuf.MdiParent = this;
      iuBuf.Show();
    }

    //buzon donde recibe mensajes
    if (oVia.buzonDos != null && oVia.buzonDos.Vista == null)
    {
      IUBuzon iuBuz = new IUBuzon(oVia.buzonDos);
      iuBuz.MdiParent = this;
      iuBuz.Show();
    }

    oVia.origen = true; //sp es origen de mensajes
    via = oVia;

    IUppYcc pYc = new IUppYcc(gp, oVia);
    pYc.MdiParent = this;
    pYc.Show();
  }
}

private void cmdPPCCExMCon_Click(object sender, EventArgs e)
{
  Via oVia = new Via(via);
  if (bU == null) bU = new BufferUno();
  oVia.objCompartido = bU;
  //oVia.tipoCompartido = TipoCompartido.BufferL;

  IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppPPyCC[3]);
  bool continuar = pre.ShowDialog(this) == DialogResult.OK;
  pre.Dispose();

  if (continuar)
  {
    if (bU.Vista == null)
    {
      IUBufferUno iuBuf = new IUBufferUno(bU);
      iuBuf.MdiParent = this;
      iuBuf.Show();
```

```
    }


    //buzon donde recibe mensajes
    if (oVia.buzon != null && oVia.buzon.Vista == null)
    {
      IUBuzon iuBuz = new IUBuzon(oVia.buzon);
      iuBuz.MdiParent = this;
      iuBuz.Show();
    }

    oVia.origen = false; //sp es destino de mensajes
    via = oVia;

    IUppYcc pYc = new IUppYcc(gp, oVia);
    pYc.MdiParent = this;
    pYc.Show();
  }
}

private void cmdProductorLim_Click(object sender, EventArgs e)
{
  Via oVia = new Via(via);
  if (bL == null) bL = new BufferLim();
  oVia.objCompartido = bL;

  IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppPPyCCx[0]);
  bool continuar = pre.ShowDialog(this) == DialogResult.OK;
  pre.Dispose();

  if (continuar)
  {
    if (bL.Vista == null)
    {
      IUBufferX iuBuf = new IUBufferX(bL);
      iuBuf.MdiParent = this;
      iuBuf.Show();
    }

    //buzon donde recibe mensajes
    if (oVia.buzonTres != null && oVia.buzonTres.Vista == null)
    {
      IUBuzon iuBuz = new IUBuzon(oVia.buzonTres);
      iuBuz.MdiParent = this;
      iuBuz.Show();
    }

    //buzon de ExM
    if (oVia.buzon != null && oVia.buzon.Vista == null)
    {
      IUBuzon iuBuz = new IUBuzon(oVia.buzon);
      iuBuz.MdiParent = this;
      iuBuz.Show();
    }

    oVia.origen = true; //sp es origen de mensajes
    via = oVia;

    IUppYccX pYcX = new IUppYccX(gp, oVia);
    pYcX.MdiParent = this;
    pYcX.Show();
```

```
    }
  }

  private void cmdConsumidorLim_Click(object sender, EventArgs e)
  {
    Via oVia = new Via(via);
    if (bL == null) bL = new BufferLim();
    oVia.objCompartido = bL;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppPPyCCx[1]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      if (bL.Vista == null)
      {
        IUBufferX iuBuf = new IUBufferX(bL);
        iuBuf.MdiParent = this;
        iuBuf.Show();
      }

      //buzon donde recibe mensajes
      if (oVia.buzonDos != null && oVia.buzonDos.Vista == null)
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzonDos);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }

      //buzon de ExM
      if (oVia.buzon != null && oVia.buzon.Vista == null)
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzon);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }

      oVia.origen = false; //sp es origen de mensajes
      via = oVia;

      IUppYccX pYcX = new IUppYccX(gp, oVia);
      pYcX.MdiParent = this;
      pYcX.Show();
    }
  }

  private void cmdProductorIli_Click(object sender, EventArgs e)
  {
    Via oVia = new Via(via);
    if (bI == null) bI = new BufferIli();
    oVia.objCompartido = bI;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppPPyCCi[0]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      if (bI.Vista == null)
      {
```

```
        IUBufferIli iuBuf = new IUBufferIli(bI);
        iuBuf.MdiParent = this;
        iuBuf.Show();
      }

      //buzon donde residen mensajes
      if (oVia.buzonDos != null && oVia.buzonDos.Vista == null)
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzonDos);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }

      //buzon de ExM
      if (oVia.buzon != null && oVia.buzon.Vista == null)
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzon);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }

      oVia.origen = true; //sp es origen de mensajes
      via = oVia;

      IUppYccI pYcI = new IUppYccI(gp, oVia);
      pYcI.MdiParent = this;
      pYcI.Show();
    }
  }

  private void cmdConsumidorIli_Click(object sender, EventArgs e)
  {
    Via oVia = new Via(via);
    if (bI == null) bI = new BufferIli();
    oVia.objCompartido = bI;

    IUpreIUsp pre = new IUpreIUsp(gp, oVia, ssppPPyCCi[1]);
    bool continuar = pre.ShowDialog(this) == DialogResult.OK;
    pre.Dispose();

    if (continuar)
    {
      if (bI.Vista == null)
      {
        IUBufferIli iuBuf = new IUBufferIli(bI);
        iuBuf.MdiParent = this;
        iuBuf.Show();
      }

      //buzon donde residen mensajes
      if (oVia.buzonDos != null && oVia.buzonDos.Vista == null)
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzonDos);
        iuBuz.MdiParent = this;
        iuBuz.Show();
      }

      //buzon de ExM
      if (oVia.buzon != null && oVia.buzon.Vista == null)
      {
        IUBuzon iuBuz = new IUBuzon(oVia.buzon);
```

```
            iuBuz.MdiParent = this;
            iuBuz.Show();
          }

          oVia.origen = false; //sp es origen de mensajes
          via = oVia;

          IUppYccI pYcI = new IUppYccI(gp, oVia);
          pYcI.MdiParent = this;
          pYcI.Show();
        }
      }
    }
}
```

////Componente: IUpreIUsp.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using GASP;
using subProcesos;

namespace IGASP
{
  public partial class IUpreIUsp : Form
  {
    GP gp;
    Via via;
    string prefijo, sufijo, s;
    int i,nBuzones;
    bool memObuf;

    public IUpreIUsp(GP sistema, Via oV, string[] elementos)
    {
      InitializeComponent();
      gp = sistema;
      via = oV;
      inicializarForm(elementos);
      txtNombre.Select();
      txtNombre.Select(txtNombre.Text.Length, 0);
    }

    void inicializarForm(string[] eles)
    {
      via.clase = eles[0];
      nBuzones = int.Parse(eles[1]);
      prefijo = "sP" + BCP.Todos;

      i = via.clase.Length > 3 ? 4 : via.clase.Length;
      sufijo = via.clase.Substring(0, i);

      txtNombre.Text = prefijo + sufijo;
      txtPrioridad.Text = "128";

      lblClase.Text = via.clase;

      memObuf = eles[2] != "0" || eles[3] != "0";
```

```
      if (!memObuf)
        gbMemoria.Visible = false;
      else
      {
        txtMemoria.Text =
((IValor)via.objCompartido).Valor.ToString();

        if (eles[2] != "0") gbMemoria.Text = "BUFFER: Valor Actual";
      }

      if (nBuzones > 0)
      {
        //llenar lista de buzones
        chlbBuzones.CheckOnClick = true;

chlbBuzones.Items.AddRange(gp.LeerBuzonesElegibles().ToArray());
      }
      else
        gbBuzones.Visible = false;
    }

    private void bnPreparar_Click(object sender, EventArgs e)
    {
      string mjeErr = "";
      int numero = 0;
      bool ver = txtNombre.Text.StartsWith(prefijo);

      //verificacion de nombre significativo hasta 10 car.
      if (ver)
      {
        if (txtNombre.Text.Length <= 10)
          via.nombre = txtNombre.Text;
        else
          via.nombre = txtNombre.Text.Substring(0, 10);
      }
      else
        mjeErr = "Nombre no empieza con '" + prefijo + "'";


      //verificacion de prioridad
      if (ver)
      {
        //maxima prioridad soft
        int n = gp.MaxP - gp.Niveles * gp.PrioridadesXnivel;
        try
        {
          numero = int.Parse(txtPrioridad.Text);
          ver = numero <= n && numero > 0;
        }
        catch
        {
          ver = false;
        }
        finally
        {
          if (ver)
            via.prioridad = numero;
          else  //error de prioridad
            mjeErr = "Prioridad NO esta entre 1 y " + n;
        }
      }
```

```
        //verificacion de buzones
        if (ver)
        {
          via.buzon = via.buzonDos = null;
          ver = chlbBuzones.CheckedIndices.Count == nBuzones;
          if (ver)
          {
            int vez = 1;
            foreach (int indice in chlbBuzones.CheckedIndices)
            {
              s = (string)chlbBuzones.Items[indice];
              s = s.Remove(0, "Buzon ".Length);
              if (vez == 1)
                via.buzon = gp.BuscarBuzon(int.Parse(s));
              else if(vez == 2)
                via.buzonDos = gp.BuscarBuzon(int.Parse(s));
              else if(vez == 3)
                via.buzonTres = gp.BuscarBuzon(int.Parse(s));

              vez++;
            }
          }
          else  //error de seleccion de buzones
            mjeErr = "Seleccione correctamente buzones...";
        }

        //verificacion de valor de memoria
        if (ver && memObuf)
        {
          try
          {
            numero = int.Parse(txtMemoria.Text);
            ((IValor)via.objCompartido).Valor = numero;
          }
          catch
          {
            ver = false;
          }
          finally
          {
            if (!ver)
              mjeErr = "Valor actual no funciona ...";
          }
        }

        if (ver)
        {
          Close();
          DialogResult = DialogResult.OK;
        }
        else //visualizacion de error
        {
          display.Text = mjeErr;
          display.ForeColor = Color.Red;
        }
      }
    }
}
```

C. Fuentes de subProcesos gestionados en C#.

//UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS
//FACULTAD DE CIENCIAS MATEMATICAS
//UNIDAD DE POST-GRADO
//MAESTRIA EN COMPUTACION E INFORMATICA
//TESIS DE GRADO: Ing. Francisco S. Aguilar V.
//FECHA: 2011
//Subsistema subProcesos – subProcesos gestionados con GASP

///Paquete: ISPP – Interfaces para subProcesos gestionados.

////Componente: IUSingular.cs – IGU para subproceso competitivo

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using GASP;

namespace subProcesos
{
  //IGU para los siguientes subprocesos:
  //Sumatoria y Factorial
  public partial class IUSingular : Form
  {
    GP gp;
    BCP proceso;
    Via via;
    string sufijo;
    delProcesoAbortado revProAbortado;
    bool procesoActivo = false;
    public IUSingular(GP sistema, Via oV)
    {
      InitializeComponent();
      gp = sistema;
      via = oV;
      sufijo = via.nombre;

      string s = "sP" + BCP.Todos;
      sufijo = sufijo.Remove(0, s.Length);

      Text = via.clase + " - Interfaz de " + via.nombre + " ?"
        + "<" + via.prioridad + ">";
      revProAbortado = new delProcesoAbortado(gp_evProAbortado);
      txtNumero.Select();
    }

    private void bnCrear_Click(object sender, EventArgs e)
    {
      bool ver = true;
      int numero = 0;
      Sumatoria objSum;
      Factorial objFac;
```

```
      Thread spEjecutar;
      try
      {
        numero = int.Parse(txtNumero.Text);
      }
      catch
      {
        ver = false;
      }

      if (ver)
      {
        bnCrear.Enabled = false;
        txtNumero.Enabled = false;
        lblResultado.Text = "";
        via.nombre = "sP" + BCP.Todos + sufijo;
        Text = via.clase + " - Interfaz de " + via.nombre
          + "<" + via.prioridad + ">";
        if (via.clase == "Sumatoria")
        {
          objSum = new Sumatoria(gp, numero);
          spEjecutar = new Thread(new ThreadStart(objSum.Ejecutar));
          objSum.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objSum.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objSum.VerDatos += new IndicaDatoString(this.manejarDatos);
          objSum.VerResultado += new
IndicaDatoInt(this.manejarResultado);
        }
        else
        {
          objFac = new Factorial(gp, numero);
          spEjecutar = new Thread(new ThreadStart(objFac.Ejecutar));
          objFac.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objFac.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objFac.VerDatos += new IndicaDatoString(this.manejarDatos);
          objFac.VerResultado += new
IndicaDatoInt(this.manejarResultado);
        }
        spEjecutar.Name = via.nombre;
        spEjecutar.Start();

        gp.evProAbortado += revProAbortado;
        procesoActivo = true;
      }
      else
      {
        display.AppendText("Ingrese numero correcto...\n");
        display.ScrollToCaret();
      }
    }

    private void IUSingular_FormClosing(object sender,
FormClosingEventArgs e)
    {
      if (procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
```

```
        procesoActivo = false;
        gp.Abortar(proceso);
      }
    }

    void gp_evProAbortado(string s)
    {
      if (via.nombre == s && procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        procesoActivo = false;
        this.BeginInvoke(new delProcesoAbortado(indicarAborto), new
Object[] { s });
      }
    }

    void manejarFlujo(string s)
    {
      this.BeginInvoke(new IndicaCodigo(displayar), new Object[] { s
});
    }

    void manejarDatos(string s)
    {
      this.BeginInvoke(new IndicaDatoString(presentarDatos), new
Object[] { s });
    }

    void manejarResultado(int sum)
    {
      gp.evProAbortado -= revProAbortado;
      procesoActivo = false;
      this.BeginInvoke(new IndicaDatoInt(mostrarResultado), new
Object[] { sum });
    }

    void indicarAborto(string nomProc)
    {
      display.AppendText("<" + nomProc + ": abortado...>\n");
      display.ScrollToCaret();
      bnCrear.Enabled = true;
      txtNumero.Enabled = true;
    }

    void displayar(string s)
    {
      if (chbFlujo.Checked)
      {
        display.AppendText(s);
        display.ScrollToCaret();
      }
    }

    void presentarDatos(string s)
    {
      if (chbDatos.Checked)
      {
        displayD.AppendText(s);
        displayD.ScrollToCaret();
      }
    }
```

```
    void mostrarResultado(int suma)
    {
      lblResultado.Text = suma.ToString();
      bnCrear.Enabled = true;
      txtNumero.Enabled = true;
      txtNumero.Select();
    }

    private void bnBorrar_Click(object sender, EventArgs e)
    {
      display.Clear();
    }

    private void bnBorrarD_Click(object sender, EventArgs e)
    {
      displayD.Clear();
    }
  }
}
```

////Componente: IUAcumulador.cs – IGU para subprocesos colaborativos autónomos

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using GASP;

namespace subProcesos
{
  //IGU para los siguientes subprocesos: Acumulador asíncrono,
  //Aculador asincrono c/demoras y Acumulador sincrono por exM.
  public partial class IUAcumulador : Form
  {
    GP gp;
    BCP proceso;
    Via via;
    MemCompartida mC;
    string sufijo;
    delProcesoAbortado revProAbortado;
    bool procesoActivo = false;
    public IUAcumulador(GP sistema, Via oV)
    {
      InitializeComponent();
      gp = sistema;
      via = oV;
      sufijo = oV.nombre;
      string s = "sP" + BCP.Todos;
      sufijo = sufijo.Remove(0, s.Length);
      Text = via.clase + " - Interfaz de " + via.nombre + " ?"
      + "<" + via.prioridad + ">";

      mC = (MemCompartida)via.objCompartido;
      if (via.buzon == null) bnBuzon.Visible = false;

      revProAbortado = new delProcesoAbortado(gp_evProAbortado);
      txtCantidad.Select();
```

```
    }

    private void bnCrear_Click(object sender, EventArgs e)
    {
      bool ver = true;
      int numero = 0;
      Acumulador objAcu;
      AcumuladorP objAcuP;
      AcumuladorExM objAcuExM;
      Thread spEjecutar;

      try
      {
        numero = int.Parse(txtCantidad.Text);
      }
      catch
      {
        ver = false;
      }

      if (ver)
      {
        bnCrear.Enabled = false;
        txtCantidad.Enabled = false;
        lblResultado.Text = "";
        via.nombre = "sP" + BCP.Todos + sufijo;
        Text = via.clase + " - Interfaz de " + via.nombre
          + "<" + via.prioridad + ">";
        if (via.clase == "Acumulador")
        {
          objAcu = new Acumulador(gp, numero,
(MemCompartida)via.objCompartido);
          spEjecutar = new Thread(new ThreadStart(objAcu.Ejecutar));
          objAcu.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objAcu.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objAcu.VerDatos += new IndicaDatoString(this.manejarDatos);
          objAcu.VerResultado += new
IndicaDatoInt(this.manejarResultado);
        }
        else if (via.clase == "AcumuladorP")
        {
          objAcuP = new AcumuladorP(gp, numero,
(MemCompartida)via.objCompartido);
          spEjecutar = new Thread(new ThreadStart(objAcuP.Ejecutar));
          objAcuP.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objAcuP.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objAcuP.VerDatos += new IndicaDatoString(this.manejarDatos);
          objAcuP.VerResultado += new
IndicaDatoInt(this.manejarResultado);
        }
        else
        {
          objAcuExM = new AcumuladorExM(gp, numero,
(MemCompartida)via.objCompartido);
          spEjecutar = new Thread(new
ThreadStart(objAcuExM.Ejecutar));
          objAcuExM.bcpRef =
```

```
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objAcuExM.buzonRef = via.buzon;
          objAcuExM.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objAcuExM.VerDatos += new
IndicaDatoString(this.manejarDatos);
          objAcuExM.VerResultado += new
IndicaDatoInt(this.manejarResultado);
        }
        spEjecutar.Name = via.nombre;
        spEjecutar.Start();

        gp.evProAbortado += revProAbortado;
        procesoActivo = true;
      }
      else
      {
        display.AppendText("Ingrese numero correcto...\n");
        display.ScrollToCaret();
      }
    }

    void gp_evProAbortado(string s)
    {
      if (via.nombre == s && procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        procesoActivo = false;
        this.BeginInvoke(new delProcesoAbortado(indicarAborto), new
Object[] { s });
      }
    }

    void manejarFlujo(string s)
    {
      this.BeginInvoke(new IndicaCodigo(displayar), new Object[] { s
});
    }

    void manejarDatos(string ds)
    {
      this.BeginInvoke(new IndicaDatoString(mostrarDatos), new
Object[] { ds });
    }

    void manejarResultado(int sum)
    {
      gp.evProAbortado -= revProAbortado;
      procesoActivo = false;
      this.BeginInvoke(new IndicaDatoInt(mostrarResultado), new
Object[] { sum });
    }

    void indicarAborto(string nomProc)
    {
      display.AppendText("<" + nomProc + ": abortado...>\n");
      display.ScrollToCaret();
      bnCrear.Enabled = true;
      txtCantidad.Enabled = true;
    }
```

```
    void displayar(string s)
    {
      if (chbFlujo.Checked)
      {
        display.AppendText(s);
        display.ScrollToCaret();
      }
    }

    void mostrarDatos(string s)
    {
      if (chbDatos.Checked)
      {
        displayD.AppendText(s);
        displayD.ScrollToCaret();
      }
    }

    void mostrarResultado(int suma)
    {
      lblResultado.Text = suma.ToString();
      bnCrear.Enabled = true;
      txtCantidad.Enabled = true;
    }

    private void bnBuzon_Click(object sender, EventArgs e)
    {
      IUBuzon iuBuz = (IUBuzon)via.buzon.Vista;
      if (iuBuz == null)
      {

        iuBuz = new IUBuzon(via.buzon);
        iuBuz.MdiParent = MdiParent;
        iuBuz.Show();
      }
      else
      {
        iuBuz.Close();
        iuBuz.Dispose();
      }
    }

    private void bnMemComp_Click(object sender, EventArgs e)
    {
      IUMemoria iuMem = (IUMemoria)mC.Vista;
      if (iuMem == null)
      {
        mC.Vista = iuMem = new IUMemoria(mC);
        iuMem.MdiParent = MdiParent;
        iuMem.Show();
      }
      else
      {
        iuMem.Close();
        iuMem.Dispose();
      }
    }

    private void IUAcumulador_FormClosing(object sender,
FormClosingEventArgs e)
    {
```

```
      if (procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        gp.Abortar(proceso);
      }
    }

    private void bnBorrar_Click(object sender, EventArgs e)
    {
      display.Clear();
    }

    private void bnBorrarD_Click(object sender, EventArgs e)
    {
      displayD.Clear();
    }
  }
}
```

////Componente: IUAcumuladorPar.cs – IGU para subproceso colaborativo
////complementario síncrono por señalización o comunicación de datos

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using GASP;

namespace subProcesos
{
  //IGU para subprocesos colaborativos complementarios síncronos
  //por mensajes de señalización o comunicación de datos
  public partial class IUAcumuladorPar : Form
  {
    GP gp;
    BCP proceso;
    Via via;
    MemCompartida mC;
    string sufijo;
    delProcesoAbortado revProAbortado;
    bool procesoActivo = false;
    public IUAcumuladorPar(GP sistema, Via oV)
    {
      InitializeComponent();
      gp = sistema;
      via = oV;
      sufijo = oV.nombre;
      string s = "sP" + BCP.Todos;
      sufijo = sufijo.Remove(0, s.Length);
      Text = via.clase + " Interfaz de " + via.nombre + " ?"
        + "<" + via.prioridad + ">";

      if (via.origen)
      {
        lblCantidad.Text = "Cuantos:";
        txtCantidad.Enabled = true;
        lblDato.Text = "Dato:";
        bnBuzon.Text = "Buzon " + via.buzonDos.BuzonId;
```

```
      }
      else
        bnBuzon.Text = "Buzon " + via.buzon.BuzonId;

      mC = (MemCompartida)via.objCompartido;
      if (mC == null) bnMemComp.Visible = false;

      revProAbortado = new delProcesoAbortado(gp_evProAbortado);

      if(via.origen) txtCantidad.Select();
    }

    private void bnCrear_Click(object sender, EventArgs e)
    {
      bool ver = true;
      int numero = 0;
      AcumuladorMSCo objSigCo;
      AcumuladorMSig objSig;
      AcumuladorMTCo objMTCo;
      AcumuladorMTD objMTD;
      Thread spEjecutar;

      if (via.origen)
      {
        //verificar datos de entrada
        try
        {
          numero = int.Parse(txtCantidad.Text);
        }
        catch
        {
          ver = false;
        }
      }

      if (ver)
      {
        bnCrear.Enabled = false;

        if (via.origen) txtCantidad.Enabled = false;
        else txtCantidad.Text = "";

        lblResultado.Text = "";

        via.nombre = "sP" + BCP.Todos + sufijo;
        Text = via.clase + " Interfaz de " + via.nombre
          + "<" + via.prioridad + ">";

        if (via.clase == "AcumuladorMSCo") //origen
        {
          objSigCo = new AcumuladorMSCo(gp, numero,
(MemCompartida)via.objCompartido);
          spEjecutar = new Thread(new ThreadStart(objSigCo.Ejecutar));
          objSigCo.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objSigCo.buzonRef = via.buzon;
          objSigCo.buzonDosRef = via.buzonDos;
          objSigCo.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objSigCo.VerDatos += new
IndicaDatoString(this.manejarDatos);
```

```
            objSigCo.VerResultado += new
IndicaDatoInt(this.manejarResultado);
            objSigCo.VerFinal += new IndicaSignal(this.manejarFinal);
          }
          else if (via.clase == "AcumuladorMTCo")//origen
          {
            objMTCo = new AcumuladorMTCo(gp, numero);
            spEjecutar = new Thread(new ThreadStart(objMTCo.Ejecutar));
            objMTCo.bcpRef =
            proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
            objMTCo.buzonRef = via.buzon;
            objMTCo.buzonDosRef = via.buzonDos;
            objMTCo.VerCodigo += new IndicaCodigo(this.manejarFlujo);
            objMTCo.VerDatos += new IndicaDatoString(this.manejarDatos);
            objMTCo.VerResultado += new
IndicaDatoInt(this.manejarResultado);
            objMTCo.VerFinal += new IndicaSignal(this.manejarFinal);
          }
          else if (via.clase == "AcumuladorMSig")//destino
          {
            objSig = new AcumuladorMSig(gp,
(MemCompartida)via.objCompartido);
            spEjecutar = new Thread(new ThreadStart(objSig.Ejecutar));
            objSig.bcpRef =
            proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
            objSig.buzonRef = via.buzon;
            objSig.buzonDosRef = via.buzonDos;
            objSig.VerCodigo += new IndicaCodigo(this.manejarFlujo);
            objSig.VerDatos += new IndicaDatoString(this.manejarDatos);
            objSig.VerDato += new IndicaDatoInt(this.manejarDato);
            objSig.VerResultado += new
IndicaDatoInt(this.manejarResultado);
            objSig.VerFinal += new IndicaSignal(this.manejarFinal);
          }
          else //if (via.clase == "AcumuladorMTD")//destino
          {
            objMTD = new AcumuladorMTD(gp);
            spEjecutar = new Thread(new ThreadStart(objMTD.Ejecutar));
            objMTD.bcpRef =
            proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
            objMTD.buzonRef = via.buzon;
            objMTD.buzonDosRef = via.buzonDos;
            objMTD.VerCodigo += new IndicaCodigo(this.manejarFlujo);
            objMTD.VerDatos += new IndicaDatoString(this.manejarDatos);
            objMTD.VerDato += new IndicaDatoInt(this.manejarDato);
            objMTD.VerResultado += new
IndicaDatoInt(this.manejarResultado);
            objMTD.VerFinal += new IndicaSignal(this.manejarFinal);
          }
          spEjecutar.Name = via.nombre;
          spEjecutar.Start();

          gp.evProAbortado += revProAbortado;
          procesoActivo = true;
        }
        else
        {
          display.AppendText("Ingrese numeros correctamente...\n");
```

```
        display.ScrollToCaret();
      }
    }

    void gp_evProAbortado(string s)
    {
      if (via.nombre == s && procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        procesoActivo = false;
        this.BeginInvoke(new delProcesoAbortado(indicarAborto), new
Object[] { s });
      }
    }

    void manejarFlujo(string s)
    {
      this.BeginInvoke(new IndicaCodigo(displayar), new Object[] { s
});
    }

    void manejarDatos(string ds)
    {
      this.BeginInvoke(new IndicaDatoString(mostrarDatos), new
Object[] { ds });
    }

    void manejarDato(int n)
    {
      this.BeginInvoke(new IndicaDatoInt(mostrarDato), new Object[] {
n });
    }

    void manejarResultado(int n)
    {
      this.BeginInvoke(new IndicaDatoInt(mostrarResultado), new
Object[] { n });
    }

    void manejarFinal()
    {
      gp.evProAbortado -= revProAbortado;
      procesoActivo = false;
      this.BeginInvoke(new IndicaSignal(finalizar));
    }

    void indicarAborto(string nomProc)
    {
      display.AppendText("<" + nomProc + ": abortado...>\n");
      display.ScrollToCaret();
      bnCrear.Enabled = true;
      if(via.origen) txtCantidad.Enabled = true;
    }

    void displayar(string s)
    {
      if (chbFlujo.Checked)
      {
        display.AppendText(s);
        display.ScrollToCaret();
      }
```

```
}

void mostrarDatos(string s)
{
  if (chbDatos.Checked)
  {
    displayD.AppendText(s);
    displayD.ScrollToCaret();
  }
}

void mostrarDato(int n)
{
  txtCantidad.Text = n.ToString();
}

void mostrarResultado(int suma)
{
  lblResultado.Text = suma.ToString();
}

void finalizar()
{
  bnCrear.Enabled = true;
  if(via.origen) txtCantidad.Enabled = true;
}

private void bnBuzon_Click(object sender, EventArgs e)
{
  IUBuzon iuBuz;
  if(via.origen) iuBuz = (IUBuzon)via.buzonDos.Vista;
  else iuBuz = (IUBuzon)via.buzon.Vista;

  if (iuBuz == null)
  {
    if (via.origen) iuBuz = new IUBuzon(via.buzonDos);
    else iuBuz = new IUBuzon(via.buzon);

    iuBuz.MdiParent = MdiParent;
    iuBuz.Show();
  }
  else
  {
    iuBuz.Close();
    iuBuz.Dispose();
  }

}

private void bnMemComp_Click(object sender, EventArgs e)
{
  IUMemoria iuMem = (IUMemoria)mC.Vista;
  if (iuMem == null)
  {
    mC.Vista = iuMem = new IUMemoria(mC);
    iuMem.MdiParent = MdiParent;
    iuMem.Show();
  }
  else
  {
    iuMem.Close();
```

```
        iuMem.Dispose();
      }
    }

    private void bnBorrar_Click(object sender, EventArgs e)
    {
      display.Clear();
    }

    private void bnBorrarD_Click(object sender, EventArgs e)
    {
      displayD.Clear();
    }

    private void IUAcumuladorPar_FormClosing(object sender,
FormClosingEventArgs e)
    {
      if (procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        gp.Abortar(proceso);
      }
    }
  }
}
```

////Componente: IUppYcc.cs – IGU para subP productor/consumidor con un buffer

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using GASP;

namespace subProcesos
{
  //IGU para subproceso productor/consumidor con un buffer a/síncrono
  public partial class IUppYcc : Form
  {
    GP gp;
    BCP proceso;
    Via via;
    int paso;
    string sufijo;
    delProcesoAbortado revProAbortado;
    bool procesoActivo = false;

    public IUppYcc(GP sistema, Via oV)
    {
      InitializeComponent();
      gp = sistema;
      via = oV;
      configurarIU();
    }

    private void configurarIU()
    {
      sufijo = via.nombre;
```

```
      string s = "sP" + BCP.Todos;
      sufijo = sufijo.Remove(0, s.Length);
      Text = via.clase + " Interfaz de " + via.nombre
        + "?<" + via.prioridad + ">";

      if (!via.origen)
      {
        lblDesde.Visible = txtDesde.Visible = false;
        lbHasta.Text = "Cuantos:";
        chbDatos.Text = "Ver Consumido";
      }

      if (via.buzon == null) bnBuzon.Visible = false;
      else
        bnBuzon.Text = "Buzon " +
        (via.origen ? via.buzonDos.BuzonId : via.buzon.BuzonId);

      revProAbortado = new delProcesoAbortado(gp_evProAbortado);
    }

    private void bnCrear_Click(object sender, EventArgs e)
    {
      bool ver = true;
      int nDesde = 0, nHasta = 0;
      Productor objProd;
      Consumidor objCons;
      ProductorExM objProdExM;
      ConsumidorExM objConsExM;
      Thread spEjecutar;
      try
      {
        if (via.origen) nDesde = int.Parse(txtDesde.Text);

        nHasta = int.Parse(txtHasta.Text);
        ver = nDesde <= nHasta;
      }
      catch
      {
        ver = false;
      }

      if (ver)
      {
        bnCrear.Enabled = false;
        if (via.origen) txtDesde.Enabled = false;
        txtHasta.Enabled = false;
        via.nombre = "sP" + BCP.Todos + sufijo;
        Text = via.clase + " Interfaz de " + via.nombre
          + "<" + via.prioridad + ">";

        if (via.clase == "Productor")
        {
          objProd = new Productor(gp, nDesde, nHasta,
via.objCompartido);
          spEjecutar = new Thread(new ThreadStart(objProd.Ejecutar));
          objProd.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objProd.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objProd.VerDatos += new IndicaDatoString(this.manejarDatos);
          objProd.VerFinal += new IndicaSignal(this.manejarFinal);
```

```
        }
        else if (via.clase == "Consumidor")
        {
          objCons = new Consumidor(gp, nHasta, via.objCompartido);
          spEjecutar = new Thread(new ThreadStart(objCons.Ejecutar));
          objCons.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objCons.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objCons.VerDatos += new IndicaDatoString(this.manejarDatos);
          objCons.VerFinal += new IndicaSignal(this.manejarFinal);
        }
        else if (via.clase == "ProductorExM")
        {
          objProdExM = new ProductorExM(gp, nDesde, nHasta,
via.objCompartido);
          spEjecutar = new Thread(new
ThreadStart(objProdExM.Ejecutar));
          objProdExM.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objProdExM.buzonRef = via.buzon;
          objProdExM.buzonDosRef = via.buzonDos;
          objProdExM.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objProdExM.VerDatos += new
IndicaDatoString(this.manejarDatos);
          objProdExM.VerFinal += new IndicaSignal(this.manejarFinal);
        }
        else  //if (via.clase == "ConsumidorExM")
        {
          objConsExM = new ConsumidorExM(gp, nHasta,
via.objCompartido);
          spEjecutar = new Thread(new
ThreadStart(objConsExM.Ejecutar));
          objConsExM.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objConsExM.buzonRef = via.buzon;
          objConsExM.buzonDosRef = via.buzonDos;
          objConsExM.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objConsExM.VerDatos += new
IndicaDatoString(this.manejarDatos);
          objConsExM.VerFinal += new IndicaSignal(this.manejarFinal);
        }

        spEjecutar.Name = via.nombre;
        spEjecutar.Start();

        gp.evProAbortado += revProAbortado;
        procesoActivo = true;
        paso = 0;
      }
      else
      {
        displayFlujo.AppendText("Ingrese numeros correctos...\n");
        displayFlujo.ScrollToCaret();
      }
    }

    void gp_evProAbortado(string s)
    {
```

```
      if (via.nombre == s && procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        procesoActivo = false;
        this.BeginInvoke(new delProcesoAbortado(indicarAborto), new
Object[] { s });
      }
    }

    void manejarFlujo(string s)
    {
      this.BeginInvoke(new IndicaCodigo(mostrarFlujo), new Object[] {
s });
    }

    void manejarDatos(string dato)
    {
      this.BeginInvoke(new IndicaDatoString(mostrarDatos), new
Object[] { dato });
    }

    void manejarFinal()
    {
      gp.evProAbortado -= revProAbortado;
      procesoActivo = false;
      this.BeginInvoke(new IndicaSignal(finalizar));
    }

    void indicarAborto(string nomProc)
    {
      displayFlujo.AppendText("<" + nomProc + ": abortado...>\n");
      displayFlujo.ScrollToCaret();
      if (via.origen) txtDesde.Enabled = true;
      txtHasta.Enabled = true;
      bnCrear.Enabled = true;
    }

    void mostrarFlujo(string s)
    {
      if (chbFlujo.Checked)
      {
        displayFlujo.AppendText(s);
        displayFlujo.ScrollToCaret();
      }
    }

    void mostrarDatos(string d)
    {
      if (chbDatos.Checked)
      {
        displayDatos.AppendText("(" + ++paso + ")" + d + "\n");
        displayDatos.ScrollToCaret();
      }
    }

    void finalizar()
    {
      if (via.origen) txtDesde.Enabled = true;
      txtHasta.Enabled = true;
      bnCrear.Enabled = true;
    }
```

```
    private void bnBuffer_Click(object sender, EventArgs e)
    {
      BufferUno bU = (BufferUno)via.objCompartido;
      IUBufferUno iuBuf = (IUBufferUno)bU.Vista;

      if (iuBuf == null)
      {
        //buffer no visualizado
        iuBuf = new IUBufferUno(bU);
        iuBuf.MdiParent = MdiParent;
        iuBuf.Show();
      }
      else
      {
        //buffer se encuentra visualizado
        iuBuf.Close();
        iuBuf.Dispose();
      }
    }

    private void bnBuzon_Click(object sender, EventArgs e)
    {
      IUBuzon iuBuz;
      if (via.origen) iuBuz = (IUBuzon)via.buzonDos.Vista;
      else iuBuz = (IUBuzon)via.buzon.Vista;

      if (iuBuz == null)
      {
        if (via.origen) iuBuz = new IUBuzon(via.buzonDos);
        else iuBuz = new IUBuzon(via.buzon);

        iuBuz.MdiParent = MdiParent;
        iuBuz.Show();
      }
      else
      {
        iuBuz.Close();
        iuBuz.Dispose();
      }
    }

    private void bnBorrarFlujo_Click(object sender, EventArgs e)
    {
      displayFlujo.Clear();
    }

    private void bnBorrarDatos_Click(object sender, EventArgs e)
    {
      displayDatos.Clear();
    }

    private void IUppYcc_FormClosing(object sender,
FormClosingEventArgs e)
    {
      if (procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        gp.Abortar(proceso);
      }
    }
```

```
  }
}
```

////Componente: IUppYccX.cs – IGU para subP productor/consumidor c/buf limitado

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using GASP;

namespace subProcesos
{
  //IGU para subproceso productor/consumidor con bufferes limitados
  public partial class IUppYccX : Form
  {
    GP gp;
    BCP proceso;
    Via via;
    int paso;
    string sufijo;
    delProcesoAbortado revProAbortado;
    bool procesoActivo = false;

    public IUppYccX(GP sistema, Via oV)
    {
      InitializeComponent();
      gp = sistema;
      via = oV;
      configurarIU();
    }

    private void configurarIU()
    {
      sufijo = via.nombre;
      string s = "sP" + BCP.Todos;
      sufijo = sufijo.Remove(0, s.Length);
      Text = via.clase + " Interfaz de " + via.nombre
        + "?<" + via.prioridad + ">";

      if (!via.origen) chbDatos.Text = "Ver Consumido";

      if (via.origen)
        bnBuzon.Text = "Buzon " + via.buzonTres.BuzonId;
      else
        bnBuzon.Text = "Buzon " + via.buzonDos.BuzonId;

      bnBuzon.Text += "," + via.buzon.BuzonId;

      revProAbortado = new delProcesoAbortado(gp_evProAbortado);
    }

    private void bnCrear_Click(object sender, EventArgs e)
    {
      bool ver = true;
      int cuantos = 0;
      ProductorLim objProdLim;
      ConsumidorLim objConsLim;
```

```
      //ProductorIli objProdExM;
      //ConsumidorIli objConsExM;
      Thread spEjecutar;
      try
      {
        cuantos = int.Parse(txtCuantos.Text);
        ver = cuantos > 0;
      }
      catch
      {
        ver = false;
      }

      if (ver)
      {
        bnCrear.Enabled = false;
        txtCuantos.Enabled = false;
        via.nombre = "sP" + BCP.Todos + sufijo;
        Text = via.clase + " Interfaz de " + via.nombre
          + "<" + via.prioridad + ">";

        if (via.clase == "ProductorLim")
        {
          objProdLim = new ProductorLim(gp, cuantos,
via.objCompartido);
          spEjecutar = new Thread(new
ThreadStart(objProdLim.Ejecutar));
          objProdLim.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objProdLim.buzonRef = via.buzon;
          objProdLim.buzonDosRef = via.buzonDos;
          objProdLim.buzonTresRef = via.buzonTres;
          objProdLim.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objProdLim.VerDatos += new
IndicaDatoString(this.manejarDatos);
          objProdLim.VerFinal += new IndicaSignal(this.manejarFinal);
        }
        else  //if (via.clase == "ConsumidorLim")
        {
          objConsLim = new ConsumidorLim(gp, cuantos,
via.objCompartido);
          spEjecutar = new Thread(new
ThreadStart(objConsLim.Ejecutar));
          objConsLim.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objConsLim.buzonRef = via.buzon;
          objConsLim.buzonDosRef = via.buzonDos;
          objConsLim.buzonTresRef = via.buzonTres;
          objConsLim.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objConsLim.VerDatos += new
IndicaDatoString(this.manejarDatos);
          objConsLim.VerFinal += new IndicaSignal(this.manejarFinal);
        }

        spEjecutar.Name = via.nombre;
        spEjecutar.Start();

        gp.evProAbortado += revProAbortado;
        procesoActivo = true;
```

```
        paso = 0;
      }
      else
      {
        displayFlujo.AppendText("Ingrese numeros correctos...\n");
        displayFlujo.ScrollToCaret();
      }
    }

    void gp_evProAbortado(string s)
    {
      if (via.nombre == s && procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        procesoActivo = false;
        this.BeginInvoke(new delProcesoAbortado(indicarAborto), new
Object[] { s });
      }
    }

    void manejarFlujo(string s)
    {
      this.BeginInvoke(new IndicaCodigo(mostrarFlujo), new Object[] {
s });
    }

    void manejarDatos(string dato)
    {
      this.BeginInvoke(new IndicaDatoString(mostrarDatos), new
Object[] { dato });
    }

    void manejarFinal()
    {
      gp.evProAbortado -= revProAbortado;
      procesoActivo = false;
      this.BeginInvoke(new IndicaSignal(finalizar));
    }

    void indicarAborto(string nomProc)
    {
      displayFlujo.AppendText("<" + nomProc + ": abortado...>\n");
      displayFlujo.ScrollToCaret();
      txtCuantos.Enabled = true;
      bnCrear.Enabled = true;
    }

    void mostrarFlujo(string s)
    {
      if (chbFlujo.Checked)
      {
        displayFlujo.AppendText(s);
        displayFlujo.ScrollToCaret();
      }
    }

    void mostrarDatos(string d)
    {
      if (chbDatos.Checked)
      {
        displayDatos.AppendText("(" + ++paso + ")" + d + "\n");
```

```
    displayDatos.ScrollToCaret();
  }
}

void finalizar()
{
  txtCuantos.Enabled = true;
  bnCrear.Enabled = true;
}

private void bnBuffer_Click(object sender, EventArgs e)
{
  BufferLim bL = (BufferLim)via.objCompartido;
  IUBufferX iuBuf = (IUBufferX)bL.Vista;

  if (iuBuf == null)
  {
    //buffer no visualizado
    iuBuf = new IUBufferX(bL);
    iuBuf.MdiParent = MdiParent;
    iuBuf.Show();
  }
  else
  {
    //buffer se encuentra visualizado
    iuBuf.Close();
    iuBuf.Dispose();
  }
}

private void bnBuzon_Click(object sender, EventArgs e)
{
  IUBuzon iuBuz, iuBuzExM;
  if (via.origen) iuBuz = (IUBuzon)via.buzonTres.Vista;
  else iuBuz = (IUBuzon)via.buzonDos.Vista;

  iuBuzExM = (IUBuzon)via.buzon.Vista;

  if (iuBuzExM == null)
  {
    if (iuBuz == null) //0 0 => 1 1
    {
      if (via.origen) iuBuz = new IUBuzon(via.buzonTres);
      else iuBuz = new IUBuzon(via.buzonDos);
      iuBuz.MdiParent = MdiParent;
      iuBuz.Show();

      iuBuzExM = new IUBuzon(via.buzon);//
      iuBuzExM.MdiParent = MdiParent;
      iuBuzExM.Show();
    }
    else //0 1 => 0 0
    {
      iuBuz.Close();
      iuBuz.Dispose();
    }
  }
  else
  {
    if (iuBuz == null) //1 0 => 1 1
    {
```

```
            if (via.origen) iuBuz = new IUBuzon(via.buzonTres);
            else iuBuz = new IUBuzon(via.buzonDos);
            iuBuz.MdiParent = MdiParent;
            iuBuz.Show();
          }
          else // 1 1 => 0 0
          {
            iuBuz.Close();
            iuBuz.Dispose();
            iuBuzExM.Close();
            iuBuzExM.Dispose();
          }
        }
      }

      private void bnBorrarFlujo_Click(object sender, EventArgs e)
      {
        displayFlujo.Clear();
      }

      private void bnBorrarDatos_Click(object sender, EventArgs e)
      {
        displayDatos.Clear();
      }

      private void IUppYccX_FormClosing(object sender,
FormClosingEventArgs e)
      {
        if (procesoActivo)
        {
          gp.evProAbortado -= revProAbortado;
          gp.Abortar(proceso);
        }
      }
    }
}
```

////Componente: IUppYccI.cs – IGU para subP productor/consumidor c/buf ilimitado

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Threading;
using GASP;

namespace subProcesos
{
  //IGU para subproceso productor/consumidor con bufferes ilimitados
  public partial class IUppYccI : Form
  {
    GP gp;
    BCP proceso;
    Via via;
    int paso;
    string sufijo;
    delProcesoAbortado revProAbortado;

    bool procesoActivo = false;
```

```
    ProductorIli objProdIli;
    ConsumidorIli objConsIli;

    public IUppYccI(GP sistema, Via oV)
    {
      InitializeComponent();
      gp = sistema;
      via = oV;
      configurarIU();
    }

    private void configurarIU()
    {
      sufijo = via.nombre;
      string s = "sP" + BCP.Todos;
      sufijo = sufijo.Remove(0, s.Length);
      Text = via.clase + " Interfaz de " + via.nombre
        + "?<" + via.prioridad + ">";

      if (!via.origen) chbDatos.Text = "Ver Consumido";

      bnBuzon.Text += " " + via.buzonDos.BuzonId;

      revProAbortado = new delProcesoAbortado(gp_evProAbortado);
    }


    private void bnCrear_Click(object sender, EventArgs e)
    {
      Thread spEjecutar;

      if (!procesoActivo)
      {
        via.nombre = "sP" + BCP.Todos + sufijo;
        Text = via.clase + " Interfaz de " + via.nombre
          + "<" + via.prioridad + ">";

        if (via.clase == "ProductorIli")
        {
          objProdIli = new ProductorIli(gp, (int)nudRidmo.Value,
via.objCompartido);
          spEjecutar = new Thread(new
ThreadStart(objProdIli.Ejecutar));
          objProdIli.bcpRef =
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objProdIli.buzonRef = via.buzon;
          objProdIli.buzonDosRef = via.buzonDos;
          objProdIli.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objProdIli.VerDatos += new
IndicaDatoString(this.manejarDatos);
          objProdIli.VerFinal += new IndicaSignal(this.manejarFinal);
        }
        else  //if (via.clase == "ConsumidorIli")
        {
          objConsIli = new ConsumidorIli(gp, (int)nudRidmo.Value,
via.objCompartido);
          spEjecutar = new Thread(new
ThreadStart(objConsIli.Ejecutar));
          objConsIli.bcpRef =
```

```
          proceso = gp.CrearProceso(spEjecutar, via.prioridad,
via.nombre);
          objConsIli.buzonRef = via.buzon;
          objConsIli.buzonDosRef = via.buzonDos;
          objConsIli.VerCodigo += new IndicaCodigo(this.manejarFlujo);
          objConsIli.VerDatos += new
IndicaDatoString(this.manejarDatos);
          objConsIli.VerFinal += new IndicaSignal(this.manejarFinal);
        }

        spEjecutar.Name = via.nombre;
        spEjecutar.Start();

        gp.evProAbortado += revProAbortado;
        paso = 0;
        bnCrear.Text = "Terminar subproceso";
        procesoActivo = true;
      }
      else
      {
        if (via.clase == "ProductorIli")
          objProdIli.Vivo = false;
        else //if (via.clase == "ConsumidorIli")
          objConsIli.Vivo = false;
      }
    }

    void gp_evProAbortado(string s)
    {
      if (via.nombre == s && procesoActivo)
      {
        gp.evProAbortado -= revProAbortado;
        procesoActivo = false;
        this.BeginInvoke(new delProcesoAbortado(indicarAborto), new
Object[] { s });
      }
    }

    void manejarFlujo(string s)
    {
      this.BeginInvoke(new IndicaCodigo(mostrarFlujo), new Object[] {
s });
    }

    void manejarDatos(string dato)
    {
      this.BeginInvoke(new IndicaDatoString(mostrarDatos), new
Object[] { dato });
    }

    void manejarFinal()
    {
      gp.evProAbortado -= revProAbortado;
      procesoActivo = false;
      this.BeginInvoke(new IndicaSignal(finalizar));
    }

    void indicarAborto(string nomProc)
    {
      displayFlujo.AppendText("<" + nomProc + ": abortado...>\n");
      displayFlujo.ScrollToCaret();
```

```
  bnCrear.Text = "Crear subproceso";
}

void mostrarFlujo(string s)
{
  if (chbFlujo.Checked)
  {
    displayFlujo.AppendText(s);
    displayFlujo.ScrollToCaret();
  }
}

void mostrarDatos(string d)
{
  if (chbDatos.Checked)
  {
    displayDatos.AppendText("(" + ++paso + ")" + d + "\n");
    displayDatos.ScrollToCaret();
  }
}

void finalizar()
{
  bnCrear.Text = "Crear subproceso";
}

private void bnBuffer_Click(object sender, EventArgs e)
{
  BufferIli bI = (BufferIli)via.objCompartido;
  IUBufferIli iuBuf = (IUBufferIli)bI.Vista;

  if (iuBuf == null)
  {
    //buffer no visualizado
    iuBuf = new IUBufferIli(bI);
    iuBuf.MdiParent = MdiParent;
    iuBuf.Show();
  }
  else
  {
    //buffer se encuentra visualizado
    iuBuf.Close();
    iuBuf.Dispose();
  }
}

private void bnBuzon_Click(object sender, EventArgs e)
{
  IUBuzon iuBuz, iuBuzExM;
  iuBuz = (IUBuzon)via.buzonDos.Vista;
  iuBuzExM = (IUBuzon)via.buzon.Vista;

  if (iuBuzExM == null)
  {
    if (iuBuz == null) //0 0 => 1 1
    {
      iuBuz = new IUBuzon(via.buzonDos);
      iuBuz.MdiParent = MdiParent;
      iuBuz.Show();

      iuBuzExM = new IUBuzon(via.buzon);//
```

```
            iuBuzExM.MdiParent = MdiParent;
            iuBuzExM.Show();
          }
          else //0 1 => 0 0
          {
            iuBuz.Close();
            iuBuz.Dispose();
          }
        }
        else
        {
          if (iuBuz == null) //1 0 => 1 1
          {
            iuBuz = new IUBuzon(via.buzonDos);
            iuBuz.MdiParent = MdiParent;
            iuBuz.Show();
          }
          else // 1 1 => 0 0
          {
            iuBuz.Close();
            iuBuz.Dispose();
            iuBuzExM.Close();
            iuBuzExM.Dispose();
          }
        }
      }

      private void nudRidmo_ValueChanged(object sender, EventArgs e)
      {
        if (procesoActivo)
        {
          if (via.clase == "ProductorIli")
            objProdIli.Ritmo = (int)nudRidmo.Value;
          else //if (via.clase == "ConsumidorIli")
            objConsIli.Ritmo = (int)nudRidmo.Value;
        }
      }

      private void bnBorrarFlujo_Click(object sender, EventArgs e)
      {
        displayFlujo.Clear();
      }

      private void bnBorrarDatos_Click(object sender, EventArgs e)
      {
        displayDatos.Clear();
      }

      private void IUppYccI_FormClosing(object sender,
  FormClosingEventArgs e)
      {
        if (procesoActivo)
        {
          gp.evProAbortado -= revProAbortado;
          gp.Abortar(proceso);
        }
      }
    }
  }
```

///Paquete: SPP – capas de aplicación de subprocesos gestionados.

////Componente: Sumatoria.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using GASP;

namespace subProcesos
{
  //Sumatoria: subproceso independiente (no colaborativo)
  public class Sumatoria : Tarea
  {
    public Sumatoria(GP gpp, int n)
      : base(gpp)
    {
      numero = n;
    }

    int numero;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoInt VerResultado;
    public event IndicaDatoString VerDatos;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerCodigo("n = 0\n");
      int n = 0;
      reloj.WaitOne();
      VerCodigo("acumulado = 0\n");
      int acumulado = 0;
      VerDatos(string.Format("S = {0}, n={1}\n",acumulado, n));
      reloj.WaitOne();
      VerCodigo("while (n < numero)\n");
      while (n < numero)
      {
        VerCodigo("(n < numero)\n");
        reloj.WaitOne();
        VerCodigo("n++\n");
        n++;
        VerDatos(string.Format("n={0}\n", n));
        reloj.WaitOne();
        VerCodigo("acumulado += n\n");
        acumulado += n;
        VerDatos(string.Format("S = {0}\n", acumulado));
        reloj.WaitOne();
      }
      VerDatos(string.Format("S = {0}\n\n", acumulado));
      VerCodigo("acumulado =>\n");
      VerResultado(acumulado);
      VerCodigo("gp.Fin()\n");
      gp.Fin();
    }
  }
}
```

////Componente: AcumuladorP.cs

```
using System;
```

```
using System.Collections.Generic;
using System.Text;
using GASP;

namespace subProcesos
{
  public class AcumuladorP : Tarea
  {
    //Acumulador asincrono con demoras (dormidas)
    //los subprocesos de este tipo acumulan en 'mComun'
    //IGU: clase IUAcumulador
    public AcumuladorP(GP gpp, int n, MemCompartida mem)
      : base(gpp)
    {
      numero = n;
      mComun = mem;
    }

    int numero;
    MemCompartida mComun;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoString VerDatos;
    public event IndicaDatoInt VerResultado;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerDatos(string.Format("n= {0}\n", numero));
      VerCodigo("Random r = new Random()\n");
      Random r = new Random();
      VerDatos(string.Format("r= {0}\n", r));
      reloj.WaitOne();
      VerCodigo("mParticular = mComun.Valor\n");
      int mParticular = mComun.Valor;
      VerDatos(string.Format("A= {0}\n", mParticular));
      reloj.WaitOne();
      VerCodigo("mParticular = mParticular + numero\n");
      VerDatos(string.Format("A({0}) + n({1})\n", mParticular,
numero));
      mParticular = mParticular + numero;
      VerDatos(string.Format("A= {0}\n", mParticular));
      reloj.WaitOne();
      VerCodigo("int cuotaDormir = r.Next(1, 5)\n");
      int cuotaDormir = r.Next(1, 5);
      reloj.WaitOne();
      VerCodigo("Dormir(cuotaDormir=" + cuotaDormir + ")\n");
      Dormir(cuotaDormir);
      VerDatos(string.Format("r= {0}\n", cuotaDormir));
      VerCodigo("mComun.Valor = mParticular\n");
      mComun.Valor = mParticular;
      reloj.WaitOne();
      VerCodigo(string.Format("A({0}) ==>\n", mParticular));
      VerResultado(mParticular);
      VerDatos(string.Format("A= {0}\n\n", mParticular));
      reloj.WaitOne();
      VerCodigo("gp.Fin()\n");
      gp.Fin();
    }
  }
}
```

////Componente: AcumuladorMSCo.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using GASP;

namespace subProcesos
{
  //Acumulador colaborativo complementario
  //sincrono por señalización con mensajes
  //este tipo de suprocesos genera 'valor' y envia por 'mComun'
  //envía señal en 'buzon' y recibe señal en 'buzonDos'
  //IGU: clase IUAcumuladorPar
  public class AcumuladorMSCo : Tarea
  {
    public AcumuladorMSCo(GP gpp, int n, MemCompartida m)
      : base(gpp)
    {
      mComun = m;
      numero = n;
    }

    MemCompartida mComun;
    int numero;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoString VerDatos;
    public event IndicaDatoInt VerResultado; //en lblResultado
    public event IndicaSignal  VerFinal;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerCodigo("Random r = new Random()\n");
  Random r = new Random();
      reloj.WaitOne();
      VerCodigo("int n = 1, valor\n");
  int n = 1, valor;
      reloj.WaitOne();
      VerCodigo("Mensaje mensaje = new Mensaje()\n");
  Mensaje mensaje = new Mensaje();
      reloj.WaitOne();
      VerCodigo("while (n <= numero)\n");
      while (n <= numero)
      {
        VerCodigo(string.Format("n({0}) <= numero({1})\n", n,
numero));
        VerCodigo("valor = r.Next(1, 100)\n");
    valor = r.Next(1, 100);
        reloj.WaitOne();
        VerCodigo("VerResultado(valor)\n");
        VerResultado(valor);
        reloj.WaitOne();
        VerCodigo("mComun.Valor = valor\n");
    mComun.Valor = valor;
        reloj.WaitOne();

        VerCodigo(string.Format("VerDatos n({0}): valor{1} =>\n", n,
valor));
        VerDatos(string.Format("n({0}): valor = {1}\n\n", n, valor));
```

```
        reloj.WaitOne();

        VerCodigo(string.Format("Enviar(buzon{0},mensaje)\n",
buzon.BuzonId));
    Enviar(buzon,mensaje);

        VerCodigo(string.Format("mensaje =
Recibir(buzon{0})\n",buzonDos.BuzonId));
    mensaje = Recibir(buzonDos);

        reloj.WaitOne();
        VerCodigo("n++\n");
    n++;
      }
      VerCodigo("mComun.Valor = -1\n");
  mComun.Valor = -1;
      VerCodigo(string.Format("Enviar(buzon{0},mensaje)\n",
buzon.BuzonId));
      Enviar(buzon, mensaje);

      VerCodigo(" VerFinal() =>\n");
      VerFinal();
      VerCodigo("gp.Fin()\n");
      gp.Fin();
    }
  }
}
```

////Componente: AcumuladorMSig.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using GASP;

namespace subProcesos
{
  //Acumulador colaborativo complementario
  //sincrono por señalización con mensajes
  //este tipo de suprocesos recibe 'valor' en 'mComun' y lo acumula
  //recibe señal en 'buzon' y envía señal en 'buzonDos'
  //IGU: clase IUAcumuladorPar
  public class AcumuladorMSig : Tarea
  {
    public AcumuladorMSig(GP gpp, MemCompartida m)
      : base(gpp)
    {
      mComun = m;
    }

    MemCompartida mComun;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoString VerDatos;
    public event IndicaDatoInt VerDato; //en txtCantidad
    public event IndicaDatoInt VerResultado;//en lblResultado
    public event IndicaSignal VerFinal;

    public override void Ejecutar()
    {
      reloj.WaitOne();
```

```
      VerCodigo("int valor, mParticular = 0\n");
  int valor, mParticular = 0, paso = 0;
      reloj.WaitOne();

      VerCodigo("Mensaje mensaje\n");
  Mensaje mensaje;
      VerCodigo("bool continuar = true\n");
  bool continuar = true;
      reloj.WaitOne();

      while (continuar)
      {
        VerCodigo(string.Format("mensaje = Recibir(buzon{0})\n",
buzon.BuzonId));
    mensaje = Recibir(buzon);
        reloj.WaitOne();

        VerCodigo("valor = mComun.Valor\n");
    valor = mComun.Valor;
        reloj.WaitOne();

        VerCodigo("continuar = valor > -1\n");
    continuar = valor > -1;
        reloj.WaitOne();

        VerCodigo("if (continuar)\n");
        if (continuar)
        {
          VerCodigo(string.Format("VerDato({0}) =>\n", valor));
          VerDato(valor);
          reloj.WaitOne();

          VerCodigo("paso++");
      paso++;
          reloj.WaitOne();

          VerCodigo("mParticular += valor\n");
          VerDatos(string.Format("p({0}): A({1}) + valor({2})\n",paso,
mParticular,valor));
      mParticular += valor;
          reloj.WaitOne();

          VerDatos(string.Format("     A = {0}\n", mParticular));
          reloj.WaitOne();

          VerCodigo(string.Format("VerResultado({0}) =>\n",
mParticular));
          VerResultado(mParticular);
          reloj.WaitOne();

          VerCodigo(string.Format("Enviar(buzon{0},mensaje)\n",
buzonDos.BuzonId));
      Enviar(buzonDos, mensaje);
        }
      }
      VerCodigo("VerFinal() =>\n");
      VerFinal();
      reloj.WaitOne();

      VerCodigo("gp.Fin()\n");
      gp.Fin();
```

```
    }
  }
}
```

////Componente: Productor.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using GASP;

namespace subProcesos
{
  //Productor asincrono (libre) con demoras(dormidas)
  //produce numeros 'desde' a 'hasta' para 'buffer'
  //IGU: clase IUppYcc
  public class Productor : Tarea
  {
    public Productor(GP gpp, int d, int h, object bl)
      : base(gpp)
    {
      desde = d;
      hasta = h;
      buffer = (BufferUno)bl;
    }

    BufferUno buffer;
    int desde, hasta;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoString VerDatos;
    public event IndicaSignal VerFinal;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerCodigo("int n = desde\n");
      int n = desde;
      VerCodigo("int cuotaDormir\n");
      int cuotaDormir;
      VerCodigo("Random r = new Random()\n");
      Random r = new Random();
      reloj.WaitOne();
      while (n <= hasta)
      {
        VerCodigo("n <= hasta\n");
        reloj.WaitOne();
        VerCodigo("buffer.Poner(n.ToString())\n");
        buffer.Poner(n);
        reloj.WaitOne();
        VerCodigo("n ==>\n");
        VerDatos(Thread.CurrentThread.Name + ": " + n.ToString());
        VerCodigo("n++\n");
        n++;
        reloj.WaitOne();
        VerCodigo("cuotaDormir = r.Next(1, 5)\n");
        cuotaDormir = r.Next(1, 5);
        reloj.WaitOne();
        VerCodigo("Dormir(cuotaDormir=" + cuotaDormir + ")\n");
        Dormir(cuotaDormir);
        reloj.WaitOne();
```

```
      }
      VerCodigo("VerFinal() =>\n");
      VerFinal();
      reloj.WaitOne();
      VerCodigo("gp.Fin()\n");
      gp.Fin();
    }
  }
}
```

////Componente: Consumidor.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using GASP;

namespace subProcesos
{
  //Consumidor asincrono (libre) con demoras(dormidas)
  //consume 'numero' numeros desde 'buffer'
  //IGU: clase IUppYcc
  public class Consumidor : Tarea
  {
    public Consumidor(GP gpp, int n, object bl)
      : base(gpp)
    {
      numero = n;
      buffer = (BufferUno)bl;
    }

    BufferUno buffer;
    int numero;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoString VerDatos;
    public event IndicaSignal VerFinal;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerCodigo("Random r = new Random()\n");
      Random r = new Random();
      reloj.WaitOne();
      VerCodigo("int n = 0, valor\n");
      int n = 0;
      VerCodigo("string dato\n");
      int dato;
      VerCodigo("int cuotaDormir\n");
      int cuotaDormir;
      reloj.WaitOne();
      while (n < numero)
      {
        VerCodigo("n <= numero\n");
        VerCodigo("dato = buffer.Sacar()\n");
        dato = buffer.Sacar();
        reloj.WaitOne();
        VerCodigo("dato==>\n");
        VerDatos(Thread.CurrentThread.Name + ": " + dato);
        reloj.WaitOne();
        VerCodigo("n++\n");
```

```
        n++;
        reloj.WaitOne();
        VerCodigo("cuotaDormir = r.Next(1, 5)\n");
        cuotaDormir = r.Next(1, 5);
        reloj.WaitOne();
        VerCodigo("Dormir(cuotaDormir=" + cuotaDormir + ")\n");
        Dormir(cuotaDormir);
        reloj.WaitOne();
      }
      VerCodigo("VerFinal() =>\n");
      VerFinal();
      reloj.WaitOne();
      VerCodigo("gp.Fin()\n");
      gp.Fin();
    }
  }
}
```

////Componente: ProductorLim.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using GASP;

namespace subProcesos
{
  //Productor sincrono por señalización con mensajes, con demoras
  //produce 'hasta' numeros para N 'buffer's
  //{INICIAL}mensaje en 'buzon' de sincroniza acceso a 'buffer's
  //{INICIAL} N mensajes en 'buzonTres' señalan 'buffer's vacíos
  //mensajes en 'buzonDos' señalan 'buffer's con datos
  //IGU: clase IUppYccX
  public class ProductorLim : Tarea
  {
    public ProductorLim(GP gpp, int h, object buf)
      : base(gpp)
    {
      hasta = h;
      buffer = (BufferLim)buf;
    }

    BufferLim buffer;
    int hasta;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoString VerDatos;
    public event IndicaSignal VerFinal;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerCodigo("Random r = new Random()\n");
      Random r = new Random();
      reloj.WaitOne();

      VerCodigo("Mensaje mjeDato, mjeExM\n");
      Mensaje mjeDato, mjeExM;
      reloj.WaitOne();

      VerCodigo("int cuotaDormir, desde = 1\n");
```

```
      int cuotaDormir, desde = 1;
      reloj.WaitOne();

      while (desde <= hasta)
      {
        VerCodigo(string.Format("while (desde({0}) <=
hasta({1}))\n",desde,hasta));

        //Captura de señal de dato vacio: capturar mjeDato
        VerCodigo("mjeDato = Recibir(buzon" + buzonTres.BuzonId +
")\n");
        mjeDato = Recibir(buzonTres);
        reloj.WaitOne();

        //Registro de dato "desde" en buffer limitado comun
        //Ingreso a zona critica: recibir mjeExM
        VerCodigo("mjeExM = Recibir(buzon" + buzon.BuzonId + ")\n");
        mjeExM = Recibir(buzon);
        reloj.WaitOne();

        //Poner el dato "desde" en buffer y visualizacion
        VerCodigo("buffer.Poner(desde)\n");
        buffer.Poner(desde);
        VerDatos(Thread.CurrentThread.Name + ": " + desde);
        reloj.WaitOne();

        //salir de zona critica: entregar mjeExM
        mjeExM = new Mensaje();
        VerCodigo("Enviar(buzon" + buzon.BuzonId + ", mjeExM)\n");
        Enviar(buzon, mjeExM);
        reloj.WaitOne();

        //Señalizar dato disponible: entregar MjeDato
        mjeDato = new Mensaje("DATO");
        VerCodigo("Enviar(buzon" + buzonDos.BuzonId + ", mjeDato)\n");
        Enviar(buzonDos, mjeDato);
        reloj.WaitOne();

        //preparacion de siguiente dato
        VerCodigo("desde++\n");
        desde++;
        reloj.WaitOne();

        VerCodigo("cuotaDormir = r.Next(1, 5)\n");
        cuotaDormir = r.Next(1, 5);
        reloj.WaitOne();

        VerCodigo("Dormir(cuotaDormir=" + cuotaDormir + ")\n");
        Dormir(cuotaDormir);
        reloj.WaitOne();
      }
      VerFinal();
      VerCodigo("gp.Fin()\n");
      gp.Fin();
    }
  }
}
```

////Componente: ConsumidorLim.cs

```
using System;
using System.Collections.Generic;
```

```
using System.Text;
using System.Threading;
using GASP;

namespace subProcesos
{
  //Consumidor sincrono por señalización con mensajes, con demoras
  //consume 'hasta' numeros desde N 'buffer's
  //{INICIAL}mensaje en 'buzon' de sincroniza acceso a 'buffer's
  //{INICIAL} N mensajes en 'buzonTres' señalan 'buffer's vacíos
  //mensajes en 'buzonDos' señalan 'buffer's con datos
  //IGU: clase IUppYccX
  public class ConsumidorLim : Tarea
  {
    public ConsumidorLim(GP gpp, int n, object buf)
      : base(gpp)
    {
      numero = n;
      buffer = (BufferLim)buf;
    }

    BufferLim buffer;
    int numero;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoString VerDatos;
    public event IndicaSignal VerFinal;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerCodigo("Random r = new Random()\n");
      Random r = new Random();
      reloj.WaitOne();

      VerCodigo("Mensaje mjeDato, mjeExM\n");
      Mensaje mjeDato, mjeExM;
      reloj.WaitOne();

      VerCodigo("int dato, cuotaDormir, n = 1\n");
      int dato, cuotaDormir, n = 1;
      reloj.WaitOne();

      while (n <= numero)
      {
        VerCodigo(string.Format("while (n({0}) <=
numero({1}))\n",n,numero));

        //Captura de señal de dato disponible para consumir: captura
de mjeDato
        VerCodigo("mjeDato = Recibir(buzon" + buzonDos.BuzonId +
")\n");
        mjeDato = Recibir(buzonDos);
        reloj.WaitOne();

        //Obtencion de dato del buffer limitado comun
        //Ingreso a zona critica: captura de mjeExM
        VerCodigo("mjeExM = Recibir(buzon" + buzon.BuzonId + ")\n");
        mjeExM = Recibir(buzon);
        reloj.WaitOne();
```

```
        //Tomar el dato del buffer
        VerCodigo("dato = buffer.Sacar()\n");
        dato = buffer.Sacar();
        VerDatos(Thread.CurrentThread.Name + ": " + dato);
        reloj.WaitOne();

        //salir de zona critica: entrega mjeExM
        VerCodigo("Enviar(buzon" + buzon.BuzonId + ", mjeExM)\n");
        mjeExM = new Mensaje();
        Enviar(buzon, mjeExM);
        reloj.WaitOne();

        //Acreditar dato consumido: entrega de mjeDato con dato vacio
        mjeDato = new Mensaje("VACIO");
        VerCodigo("Enviar(buzon" + buzonTres.BuzonId + ",
mjeDato)\n");
        Enviar(buzonTres, mjeDato);
        reloj.WaitOne();

        VerCodigo("n++\n");
        n++;
        reloj.WaitOne();

        VerCodigo("cuotaDormir = r.Next(1, 5)\n");
        cuotaDormir = r.Next(1, 5);
        reloj.WaitOne();

        VerCodigo("Dormir(cuotaDormir=" + cuotaDormir + ")\n");
        Dormir(cuotaDormir);
        reloj.WaitOne();
      }

      VerFinal();
      VerCodigo("gp.Fin()\n");
      gp.Fin();
    }
  }
}
```

////Componente: ProductorIli.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using GASP;

namespace subProcesos
{
  //Productor sincrono por comunicación de datos en mjes, c/demoras
  //genera numeros aleatorios y los envía al 'buzonDos', equivale a
  //producir numeros para infinitos bufferes
  //{INICIAL}mensaje en 'buzon' sincroniza 'buffer's, redundante
  //IGU: clase IUppYccI
  public class ProductorIli : Tarea
  {
    public ProductorIli(GP gpp, int rit, object buf)
      : base(gpp)
    {
      buffer = (BufferIli)buf;
      ritmo = rit;
    }
```

```
    BufferIli buffer;
    int ritmo;
    bool vivo = true;

    public event IndicaCodigo VerCodigo;
    public event IndicaDatoString VerDatos;
    public event IndicaSignal VerFinal;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerCodigo("Random r = new Random()\n");
      Random r = new Random();
      reloj.WaitOne();

      VerCodigo("Mensaje mjeDato, mjeExM\n");
      Mensaje mjeDato, mjeExM;
      reloj.WaitOne();

      VerCodigo("int cuotaDormir\n");
      int cuotaDormir;
      reloj.WaitOne();

      while (vivo)
      {
        VerCodigo(string.Format("while (vivo({0}))\n",vivo));

        //Preparar elemento
        VerCodigo(string.Format("cuotaDormir = r.Next(1,
{0})\n",ritmo));
        cuotaDormir = r.Next(1, ritmo);
        reloj.WaitOne();

        VerCodigo("Dormir(cuotaDormir=" + cuotaDormir + ")\n");
        Dormir(cuotaDormir);
        reloj.WaitOne();

        //generar numero para mjeDato
        VerCodigo("int numero = r.Next(1, 1000)\n");
        int numero = r.Next(1, 1000);
        reloj.WaitOne();

        //armar mjeDato
        VerCodigo("mjeDato = new Mensaje(numero.ToString())\n");
        mjeDato = new Mensaje(numero.ToString());
        reloj.WaitOne();

        //Enviar mjeDato con dato y visualizacion
        VerCodigo("Enviar(buzon" + buzonDos.BuzonId + ", mjeDato)\n");
        Enviar(buzonDos, mjeDato);
        VerDatos(Thread.CurrentThread.Name + ": " + numero);
        reloj.WaitOne();

        //Respaldo de numero en buffer infinito comumn
        //Ingreso a zona critica: captura de mjeExM
        VerCodigo("mjeExM = Recibir(buzon" + buzon.BuzonId + ")\n");
        mjeExM = Recibir(buzon);
        reloj.WaitOne();

        //Respalda numero en buffer
```

```
          VerCodigo("buffer.Poner(numero)\n");
          buffer.Poner(numero);
          reloj.WaitOne();

          //salir de zona critica: entrega de mjeExM
          mjeExM = new Mensaje();
          VerCodigo("Enviar(buzon" + buzon.BuzonId + ", mjeExM)\n");
          Enviar(buzon, mjeExM);
          reloj.WaitOne();
        }
        VerFinal();
        VerCodigo("gp.Fin()\n");
        gp.Fin();
      }

      public bool Vivo
      {
        set
        {
          vivo = value;
        }
      }

      public int Ritmo
      {
        set
        {
          ritmo = value;
        }
      }
    }
}
```

////Componente: ConsumidorIli.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using GASP;

namespace subProcesos
{
  //Consumidor sincrono por comunicación de datos en mjes, c/demoras
  //recibe mensajes c/datos existentes en 'buzonDos', equivale a
  //consumir datos existentes desde 'buffer's
  //{INICIAL}mensaje en 'buzon' sincroniza 'buffer's, redundante
  //IGU: clase IUppYccI
  public class ConsumidorIli : Tarea
  {
    public ConsumidorIli(GP gpp, int rit, object buf)
      : base(gpp)
    {
      buffer = (BufferIli)buf;
      ritmo = rit;
    }

    BufferIli buffer;
    int ritmo;
    bool vivo = true;

    public event IndicaCodigo VerCodigo;
```

```
    public event IndicaDatoString VerDatos;
    public event IndicaSignal VerFinal;

    public override void Ejecutar()
    {
      reloj.WaitOne();
      VerCodigo("Random r = new Random()\n");
      Random r = new Random();
      reloj.WaitOne();

      VerCodigo("Mensaje mjeDato, mjeExM\n");
      Mensaje mjeDato, mjeExM;
      reloj.WaitOne();

      VerCodigo("int dato, datoVer, cuotaDormir\n");
      int dato, datoVer, cuotaDormir;
      reloj.WaitOne();

      while (vivo)
      {
        VerCodigo(string.Format("while (vivo({0}))\n",vivo));

        VerCodigo(string.Format("cuotaDormir = r.Next(1, {0})\n",
ritmo));
        cuotaDormir = r.Next(1, ritmo);
        reloj.WaitOne();

        VerCodigo("Dormir(cuotaDormir=" + cuotaDormir + ")\n");
        Dormir(cuotaDormir);
        reloj.WaitOne();

        //Recibir el mjeDato
        VerCodigo("mjeDato = Recibir(buzon" + buzonDos.BuzonId +
")\n");
        mjeDato = Recibir(buzonDos);
        reloj.WaitOne();

        try
        {
          //depuracion de dato y su visualizacion
          VerCodigo("dato = int.Parse((string)mjeDato.Cuerpo)\n");
          dato = int.Parse((string)mjeDato.Cuerpo);
          VerDatos(Thread.CurrentThread.Name + ": " + dato);
          reloj.WaitOne();

          //Eliminacion de respaldo de dato del buffer ilimitado comun
          //ingreso a zona critica: captura de mjeExM
          VerCodigo("mjeExM = Recibir(buzon" + buzon.BuzonId + ")\n");
          mjeExM = Recibir(buzon);
          reloj.WaitOne();

          //obtencion del dato desde el buffer y verificacion
          VerCodigo("datoVer = buffer.Sacar()\n");
          datoVer = buffer.Sacar();
          VerCodigo(string.Format("dato{0} =? datoVer{1}\n",dato,
datoVer));
          reloj.WaitOne();

          //salida de zona critica: entrega mjeExM
          mjeExM = new Mensaje();
          VerCodigo("Enviar(buzon" + buzon.BuzonId + ", mjeDato)\n");
```

```
            Enviar(buzon, mjeExM);
            reloj.WaitOne();
          }
          catch
          {
            VerCodigo("vivo = false, en catch\n");
            vivo = false;
            reloj.WaitOne();
          }
        }
        VerFinal();

        VerCodigo("gp.Fin()\n");
        gp.Fin();
      }

      public bool Vivo
      {
        set
        {
          vivo = value;
        }
      }

      public int Ritmo
      {
        set
        {
          ritmo = value;
        }
      }
    }
}
```